OFFICE OF THE DIRECTOR OF NATIONAL INTELLIGENCE

STONESOUP

Securely Taking On Software of Uncertain Provenance

Intelligence Advanced Research Projects Activity

LEADING INTELLIGENCE INTEGRATION

STONESOUP

Test Case Weakness Variants (Snippets) Documentation

5 August 2014

This report was prepared by TASC, Inc., Ponte Technologies LLC, and i_SW LLC. Supported by the Intelligence Advanced Research Projects Activity (IARPA), Research Operational Support Environment (ROSE) contract number 2011-110902-00005-002.
The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation hereon. Disclaimer: The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of IARPA or the U.S. Government.

Approved for public release; distribution unlimited.

Table of Contents

1. Weakness Documentation ... 6
   1.1 Restrictions and Requirements ... 6
   1.2 Execution Environment ... 7
   1.3 Concurrency Handling ... 8
      1.3.1 Multi-Process Race Conditions ... 10
      1.3.2 Multi-Threaded Race Conditions ... 13
      1.3.3 C - CWE-363A - Race Condition Enabling Link Following ... 17
      1.3.4 C - CWE-367A - Time-of-check Time-of-use (TOCTOU) Race Condition ... 21
      1.3.5 C - CWE-412A - Unrestricted Externally Accessible Lock ... 25
      1.3.6 C - CWE-414A - Missing Lock Check ... 27
      1.3.7 C - CWE-479A - Signal Handler Use of a Non-reentrant Function ... 31
      1.3.8 C - CWE-543A - Use of Singleton Pattern Without Synchronization in a Multithreaded Context ... 35
      1.3.9 C - CWE-609A - Double-Checked Locking ... 39
      1.3.10 C - CWE-663A - Use of a Non-reentrant Function in a Concurrent Context ... 43
      1.3.11 C - CWE-764A - Multiple Locks of a Critical Resource ... 46
      1.3.12 C - CWE-765A - Multiple Unlocks of a Critical Resource ... 48
      1.3.13 C - CWE-765B - Multiple Unlocks of a Critical Resource ... 51
      1.3.14 C - CWE-820A - Missing Synchronization ... 55
      1.3.15 C - CWE-821A - Incorrect Synchronization ... 59
      1.3.16 C - CWE-828A - Signal Handler with Functionality that is not Asynchronous-Safe ... 62
      1.3.17 C - CWE-831A - Signal Handler Function Associated with Multiple Signals ... 66
      1.3.18 C - CWE-833A - Deadlock ... 69
      1.3.19 J - CWE-363A - Race Condition Enabling Link Following ... 74
      1.3.20 J - CWE-367A - Time-of-check Time-of-use (TOCTOU) Race Condition ... 78
      1.3.21 J - CWE-412A - Unrestricted Externally Accessible Lock ... 81
      1.3.22 J - CWE-414A - Missing Lock Check ... 83
      1.3.23 J - CWE-543A - Use of Singleton Pattern Without Synchronization in a Multithreaded Context ... 86
      1.3.24 J - CWE-567A - Unsynchronized Access to Shared Data in a Multithreaded Context ... 91
      1.3.25 J - CWE-572A - Call to Thread run() instead of start() ... 94
      1.3.26 J - CWE-609A - Double-Checked Locking ... 96
      1.3.27 J - CWE-663A - Use of a Non-reentrant Function in a Concurrent Context ... 101
      1.3.28 J - CWE-764A - Multiple Locks of a Critical Resource ... 104
      1.3.29 J - CWE-765A - Multiple Unlocks of a Critical Resource ... 106
      1.3.30 J - CWE-820A - Missing Synchronization ... 109
      1.3.31 J - CWE-821A - Incorrect Synchronization ... 112
      1.3.32 J - CWE-832A - Unlock of a Resource that is not Locked ... 116
      1.3.33 J - CWE-833A - Deadlock ... 118
   1.4 Error Handling ... 120
      1.4.1 J - CWE-209A - Information Exposure Through An Error Message ... 121
      1.4.2 J - CWE-248A - Uncaught Exception ... 123
      1.4.3 J - CWE-252A - Unchecked Return ... 124
      1.4.4 J - CWE-252B - Unchecked Return ... 126
      1.4.5 J - CWE-253A - Incorrect Check Of Function Return Value ... 128
      1.4.6 J - CWE-390A - Detection of Error Condition Without Action ... 130
      1.4.7 J - CWE-391A - Unchecked Error Condition ... 131
      1.4.8 J - CWE-460A - Improper Cleanup On Thrown Exception ... 133
      1.4.9 J - CWE-584A - Return Inside Finally ... 135
   1.5 Memory Corruption ... 137
      1.5.1 C - CWE-120A - Stack Buffer Overflow ... 138
      1.5.2 C - CWE-120B - Heap Buffer Overflow ... 140
      1.5.3 C - CWE-120C - Heap Buffer Overflow within Struct ... 143
      1.5.4 C - CWE-120D - Stack Buffer Overflow within Struct ... 145
      1.5.5 C - CWE-124A - Buffer Underwrite ('Buffer Underflow') ... 147
      1.5.6 C - CWE-124B - Buffer Underwrite ('Buffer Underflow') ... 150
      1.5.7 C - CWE-124C - Buffer Underwrite ('Buffer Underflow') ... 152
      1.5.8 C - CWE-124D - Buffer Underwrite ('Buffer Underflow') ... 155
      1.5.9 C - CWE-126A - Buffer Over-Read ... 157
      1.5.10 C - CWE-126B - Buffer Over-Read ... 159
      1.5.11 C - CWE-126C - Buffer Over-Read ... 161
      1.5.12 C - CWE-126D - Buffer Over-Read ... 164
      1.5.13 C - CWE-127A - Buffer Under-Read ... 166
      1.5.14 C - CWE-127B - Buffer Under-Read ... 169
      1.5.15 C - CWE-127C - Buffer Under-Read ... 170
      1.5.16 C - CWE-127D - Buffer Under-Read ... 173
      1.5.17 C - CWE-129A - Improper Validation of Array Index ... 175
      1.5.18 C - CWE-129B - Improper Validation of Array Index ... 178

Approved for public release; distribution unlimited.

1.5.19 C - CWE-134A - Uncontrolled Format String . . . 180
1.5.20 C - CWE-134B - Uncontrolled Format String . . . 182
1.5.21 C - CWE-170A - Improper Null Termination . . . 184
1.5.22 C - CWE-170B - Improper Null Termination . . . 186
1.5.23 C - CWE-415A - Double Free . . . 188
1.5.24 C - CWE-416A - Use After Free . . . 191
1.5.25 C - CWE-590A - Free of Memory not on the Heap . . . 192
1.5.26 C - CWE-590B - Free of Memory not on the Heap . . . 194
1.5.27 C - CWE-761A - Free of Memory not at Start of Buffer . . . 196
1.5.28 C - CWE-785A - Use of Path Manipulation Function without Maximum-sized Buffer on the Stack . . . 198
1.5.29 C - CWE-785B - Use of Path Manipulation Function without Maximum-sized Buffer on the Heap . . . 200
1.5.30 C - CWE-785C - Use of Path Manipulation Function without Maximum-sized Buffer on the Heap in a Struct . . . 202
1.5.31 C - CWE-785D - Use of Path Manipulation Function without Maximum-sized Buffer on the Stack in a Struct . . . 204
1.5.32 C - CWE-805A - Buffer Access With Incorrect Length Value on the Stack . . . 206
1.5.33 C - CWE-805B - Buffer Access With Incorrect Length Value on the Heap . . . 208
1.5.34 C - CWE-805C - Buffer Access With Incorrect Length Value on the Heap in a Struct . . . 210
1.5.35 C - CWE-805D - Buffer Access With Incorrect Length Value on the Stack in a Struct . . . 212
1.5.36 C - CWE-806A - Buffer Access Using Size of Source Buffer on the Heap . . . 214
1.5.37 C - CWE-806B - Buffer Access Using Size of Source Buffer on the Heap . . . 216
1.5.38 C - CWE-806C - Buffer Access Using Size of Source Buffer on the Heap in a Struct . . . 219
1.5.39 C - CWE-806D - Buffer Access Using Size of Source Buffer on the Stack in a Struct . . . 221
1.5.40 C - CWE-822A - Untrusted Pointer Dereference . . . 224
1.5.41 C - CWE-824A - Access of Uninitialized Pointer on the Stack . . . 226
1.5.42 C - CWE-824B - Access of Uninitialized Pointer on the Heap . . . 228
1.5.43 C - CWE-843A - Access of Resource Using Incompatible Type ('Type Confusion') . . . 230
1.6 Null Pointer Errors . . . 232
1.6.1 C - CWE-476A - NULL Pointer Dereference . . . 232
1.6.2 C - CWE-476B - NULL Pointer Dereference . . . 235
1.6.3 C - CWE-476C - NULL Pointer Dereference . . . 237
1.6.4 C - CWE-476D - NULL Pointer Dereference . . . 240
1.6.5 C - CWE-476E - NULL Pointer Dereference . . . 242
1.6.6 C - CWE-476F - NULL Pointer Dereference . . . 244
1.6.7 C - CWE-476G - NULL Pointer Dereference . . . 246
1.7 Number Handling . . . 247
1.7.1 C - CWE-190A - Integer Overflow or Wraparound . . . 248
1.7.2 C - CWE-191A - Integer Underflow (Wrap or Wraparound) . . . 250
1.7.3 C - CWE-191B - Integer Underflow (Wrap or Wraparound) . . . 252
1.7.4 C - CWE-194A - Unexpected Sign Extention . . . 254
1.7.5 C - CWE-195A - Signed to Unsigned Conversion Error . . . 257
1.7.6 C - CWE-196A - Unsigned to Signed Conversion Error . . . 259
1.7.7 C - CWE-197A - Numeric Truncation Error . . . 260
1.7.8 C - CWE-369A - Divide By Zero . . . 263
1.7.9 C - CWE-682A - Incorrect Calculation . . . 264
1.7.10 C - CWE-682B - Incorrect Calculation . . . 267
1.7.11 C - CWE-839A - Divide By Zero . . . 269
1.7.12 J - CWE-190A - Integer Overflow or Wraparound . . . 270
1.7.13 J - CWE-190B - Integer Overflow or Wraparound . . . 272
1.7.14 J - CWE-191A - Integer Underflow (Wrap or Wraparound) . . . 273
1.7.15 J - CWE-194A - Unexpected Sign Extention . . . 275
1.7.16 J - CWE-195A - Signed to Unsigned Conversion Error . . . 277
1.7.17 J - CWE-196A - Unsigned to Signed Conversion Error . . . 279
1.7.18 J - CWE-197A - Numeric Truncation Error . . . 280
1.7.19 J - CWE-369A - Divide by Zero . . . 282
1.7.20 J - CWE-839A - Numeric Range Comparison Without Minimum Check . . . 283
1.8 Appendix A: Weakness Source Code . . . 284
1.8.1 C Weakness Source Code . . . 286
1.8.1.1 C - CWE-120A Source Code . . . 287
1.8.1.2 C - CWE-120B Source Code . . . 291
1.8.1.3 C - CWE-120C Source Code . . . 294
1.8.1.4 C - CWE-120D Source Code . . . 297
1.8.1.5 C - CWE-124A Source Code . . . 300
1.8.1.6 C - CWE-124B Source Code . . . 303
1.8.1.7 C - CWE-124C Source Code . . . 306
1.8.1.8 C - CWE-124D Source Code . . . 310
1.8.1.9 C - CWE-126A Source Code . . . 313
1.8.1.10 C - CWE-126B Source Code . . . 316
1.8.1.11 C - CWE-126C Source Code . . . 319
1.8.1.12 C - CWE-126D Source Code . . . 322
1.8.1.13 C - CWE-127A Source Code . . . 325
1.8.1.14 C - CWE-127B Source Code . . . 328
1.8.1.15 C - CWE-127C Source Code . . . 331
1.8.1.16 C - CWE-127D Source Code . . . 334

1.8.1.17 C - CWE-129A Source Code 337
1.8.1.18 C - CWE-129B Source Code 341
1.8.1.19 C - CWE-134A Source Code 345
1.8.1.20 C - CWE-134B Source Code 348
1.8.1.21 C - CWE-170A Source Code 351
1.8.1.22 C - CWE-170B Source Code 354
1.8.1.23 C - CWE-190A Source Code 357
1.8.1.24 C - CWE-191A Source Code 360
1.8.1.25 C - CWE-191B Source Code 363
1.8.1.26 C - CWE-194A Source Code 366
1.8.1.27 C - CWE-195A Source Code 370
1.8.1.28 C - CWE-196A Source Code 373
1.8.1.29 C - CWE-197A Source Code 376
1.8.1.30 C - CWE-363A Source Code 379
1.8.1.31 C - CWE-367A Source Code 385
1.8.1.32 C - CWE-369A Source Code 391
1.8.1.33 C - CWE-412A Source Code 394
1.8.1.34 C - CWE-414A Source Code 397
1.8.1.35 C - CWE-415A Source Code 403
1.8.1.36 C - CWE-416A Source Code 406
1.8.1.37 C - CWE-476A Source Code 410
1.8.1.38 C - CWE-476B Source Code 413
1.8.1.39 C - CWE-476C Source Code 416
1.8.1.40 C - CWE-476D Source Code 420
1.8.1.41 C - CWE-476E Source Code 426
1.8.1.42 C - CWE-476F Source Code 429
1.8.1.43 C - CWE-476G Source Code 432
1.8.1.44 C - CWE-479A Source Code 435
1.8.1.45 C - CWE-543A Source Code 441
1.8.1.46 C - CWE-590A Source Code 448
1.8.1.47 C - CWE-590B Source Code 451
1.8.1.48 C - CWE-609A Source Code 454
1.8.1.49 C - CWE-663A Source Code 460
1.8.1.50 C - CWE-682A Source Code 466
1.8.1.51 C - CWE-682B Source Code 470
1.8.1.52 C - CWE-761A Source Code 473
1.8.1.53 C - CWE-764A Source Code 476
1.8.1.54 C - CWE-765A Source Code 480
1.8.1.55 C - CWE-765B Source Code 487
1.8.1.56 C - CWE-785A Source Code 493
1.8.1.57 C - CWE-785B Source Code 496
1.8.1.58 C - CWE-785C Source Code 499
1.8.1.59 C - CWE-785D Source Code 502
1.8.1.60 C - CWE-805A Source Code 505
1.8.1.61 C - CWE-805B Source Code 508
1.8.1.62 C - CWE-805C Source Code 511
1.8.1.63 C - CWE-805D Source Code 514
1.8.1.64 C - CWE-806A Source Code 517
1.8.1.65 C - CWE-806B Source Code 520
1.8.1.66 C - CWE-806C Source Code 523
1.8.1.67 C - CWE-806D Source Code 526
1.8.1.68 C - CWE-820A Source Code 529
1.8.1.69 C - CWE-821A Source Code 535
1.8.1.70 C - CWE-822A Source Code 541
1.8.1.71 C - CWE-824A Source Code 544
1.8.1.72 C - CWE-824B Source Code 547
1.8.1.73 C - CWE-828A Source Code 551
1.8.1.74 C - CWE-831A Source Code 557
1.8.1.75 C - CWE-833A Source Code 563
1.8.1.76 C - CWE-839A Source Code 569
1.8.1.77 C - CWE-843A Source Code 572
1.8.2 Java Weakness Source Code 575
1.8.2.1 J - CWE-190A Source Code 575
1.8.2.2 J - CWE-190B Source Code 578
1.8.2.3 J - CWE-191A Source Code 580
1.8.2.4 J - CWE-194A Source Code 582
1.8.2.5 J - CWE-195A Source Code 584
1.8.2.6 J - CWE-196A Source Code 586
1.8.2.7 J - CWE-197A Source Code 588
1.8.2.8 J - CWE-209A Source Code 590
1.8.2.9 J - CWE-248A Source Code 593
1.8.2.10 J - CWE-252A Source Code 595

1.8.2.11 J - CWE-252B Source Code . . . 597
1.8.2.12 J - CWE-253A Source Code . . . 599
1.8.2.13 J - CWE-363A Source Code . . . 601
1.8.2.14 J - CWE-367A Source Code . . . 604
1.8.2.15 J - CWE-369A Source Code . . . 607
1.8.2.16 J - CWE-390A Source Code . . . 609
1.8.2.17 J - CWE-391A Source Code . . . 611
1.8.2.18 J - CWE-412A Source Code . . . 613
1.8.2.19 J - CWE-414A Source Code . . . 615
1.8.2.20 J - CWE-460A Source Code . . . 619
1.8.2.21 J - CWE-543A Source Code . . . 621
1.8.2.22 J - CWE-567A Source Code . . . 626
1.8.2.23 J - CWE-572A Source Code . . . 630
1.8.2.24 J - CWE-584A Source Code . . . 634
1.8.2.25 J - CWE-609A Source Code . . . 637
1.8.2.26 J - CWE-663A Source Code . . . 642
1.8.2.27 J - CWE-764A Source Code . . . 647
1.8.2.28 J - CWE-765A Source Code . . . 650
1.8.2.29 J - CWE-820A Source Code . . . 653
1.8.2.30 J - CWE-821A Source Code . . . 658
1.8.2.31 J - CWE-832A Source Code . . . 663
1.8.2.32 J - CWE-833A Source Code . . . 666
1.8.2.33 J - CWE-839A Source Code . . . 669
1.9 Appendix B: Supporting Scripts . . . 671
1.9.1 runFifos.py . . . 671

Weakness Documentation

STONESOUP develops and demonstrates comprehensive, automated techniques that allow end users to securely execute software without basing risk mitigations on characteristics of provenance that have a dubious relationship to security. Existing techniques to find and remove software vulnerabilities are costly, labor-intensive, and time-consuming. Many risk management decisions are therefore based on qualitative and subjective assessments of the software suppliers' trustworthiness. STONESOUP develops software analysis, confinement, and diversification techniques so that non-experts can transform questionable software into more secure versions without changing the behavior of the programs. STONESOUP research is currently focused on mitigating weaknesses in C, Java, and pre-compiled Binary applications.

A core component of the STONESOUP program includes the Test & Evaluation (T&E) of proposed technologies to evaluate their effectiveness in mitigating or stopping exploitation of many common classes of software weaknesses. To this end, a number of software snippets were developed to provide discrete tests of specific weaknesses. These individual weakness variants are extremely simplistic, and in many cases demonstrate the core weakness, performing no further meaningful processing. These weakness variants form the basis form which a larger test suite is generated.

Another evaluation metric is the scalability of each proposed technology with respect to handling larger code bases, determined by a Lines of Code (LOC) count. To support this testing, the weakness variants are seeded/injected into larger base programs.

Restrictions and Requirements

The IARPA STONESOUP Program seeks solutions that substantially reduce the malicious exploitation of software vulnerabilities by (1) extending the scope and capability of the analysis, confinement, and diversification techniques; (2) addressing a wide range of security vulnerabilities within the same framework; and (3) integrating analysis, confinement, and diversification to leverage the strengths and weaknesses of each technique. The desired major advancement sought is to provide comprehensive, automated techniques for vulnerability reduction in software of uncertain provenance.

Designing and implementing software weaknesses that provide valid and fair tests in the spirit of the STONESOUP Program is a challenging task. There are a number of requirements that must be met or considered when designed each weakness algorithmic variant.

Application Specific Knowledge

One of the core principles of the STONESOUP program is that proposed technologies should not require anything such as a functional specification of the applications being protected. As a result, a weakness algorithmic variant must be a fundamental flaw in the design or implementation of a representative piece of software, devoid of application specific logic. While the intent of this requirement is straight forward, there is no clear measurement to determine if an implemented algorithmic variant for a target weakness has stayed too far into the application specific knowledge domain. Determining if a weakness algorithmic variant requires application specific knowledge is a completely subjective measure, one that could result in endless debate over the fairness of the test in question.

Instead, a set of guidelines has been used to aid in evaluating if a weakness algorithmic variant requires application specific knowledge to properly mitigate the negative technical impact.

1. Is the software implementation of the weakness performing a set of operations that could not be expected in a fundamentally different processing context?
2. Does the execution of a set of inputs require a custom policy to distinguish good from bad?

If both of these evaluations are true, the implementation of the weakness algorithmic variant may be too application specific, and requires a re-design to allow for a proposed technology to identify and mitigate.

Negative Technical Impacts

There are a number of software defects that may not be considered exploitable simply because there is direct negative technical impact. In addition, if there is not direct negative technical impact from trigger a software weakness, proposed technologies may have a decide to not take corrective or preventative actions. As part of weakness algorithmic variant design and to support Test & Evaluation, each weakness algorithmic variant must map to one or more negative technical impacts as defined below.

- Confidentiality
    - Read Files or Directories
    - Read Application Data
- Integrity of Origin
    - Gain Privileges/Assume Identity
    - [Hide Activities]
    - Execute Unauthorized Code or Commands
- Integrity of Data

  - Modify Files or Directories
  - Modify Application Data
- Integrity of Application
  - Bypass Protection Mechanism
  - [Alter Execution Logic]
  - [Unexpected State]
- Availability
  - DoS: Uncontrolled Exit (Crash)
  - [DoS: Amplification]
  - [DoS: Instability]
  - [DoS: Blocking]
  - DoS: Resource Consumption

Altered Functionality

As an artifact of the program goal to provide comprehensive, automated techniques for vulnerability reduction in software of uncertain provenance, protected/hardened software must still function normally and without modification when processing benign inputs. The proposed technologies are to only intervene and mitigate a vulnerability when an exploiting input would otherwise result in a negative technical impact. In order to support an evaluation measurement of altered functionality, each weakness algorithmic variant must support both a "good" and "bad" execution path that are triggered by benign and exploit inputs. While it would be much simpler to leave the vulnerable portion of the code in a non-exercised execution branch, this would not test the proposed technology's ability to preserve the core functionality when the software is used as intended.

Policies

Weakness algorithmic variants should be designed in a manner that an individual policy is not required for the proposed technology to understand when an input has triggered a negative technical impact. There are of course a few exceptions to this rule when the T&E Rules of Engagement establish a broader set of inferred policies.

Tainted Data

Many of the weaknesses targeted as part of the Tainted Data weakness class deal directly with path manipulation (e.g. Absolute Path Attacks and Relative Path Attacks) vulnerabilities. After careful evaluation, it was decided early in STONESOUP Phase 2 that "untrusted" input sources needed to be restricted in this context to understand when it was not desirable to permit a file to be written or read. Due to the fact that the operating system enforces user level file permissions, it is not possible to exploit yourself (unless running a setuid program). Instead, only inputs which originate from an external source should be considered "untrusted".

Resource Drains

Due to the fact that test execution software runs on the same box as the test cases during T&E, unrestricted resource drains would create difficulties in collecting observables and measuring success of a exploiting input. To this end, weakness algorithmic variants that target resource drains are allowed to establish ulimits for the process. The proposed technologies are not to use the ulimit to enforce resource restrictions, but instead ensure that the executing software does to reach this threshold during execution.

Note on Code Excerpts

Please note that when excerpts of code are provided in the weakness documentation, calls to LTTng may be omitted to improve readability. The LTTng calls do not alter the functionality of the code, but including them often obscures the intent of the programmer.

For completeness, the LTTng calls are included in the full weakness snippets provided under Weakness Source Code.

Execution Environment

All weaknesses developed for STONESOUP target a x86_64 Linux operating environment. Furthermore, proposed technologies under Test & Evaluation only targeted two specific Linux distributions. As such, each weakness variant has been more thoroughly tested and validated in these environments and with specific compiler versions. While compilation and execution on other platforms and Linux distributions may work, even subtle differences introduced in new compiler versions may change the behavior of each weakness.

Supported Linux Distributions

Weakness variants are tested and debugged on the following distributions of Linux.

- CentOS 6.5 (x86_64)
- Ubuntu 12.04 (x86_64)

Supported Compilers

C Weakness variants are tested and validated using the GNU Compiler Collection (GCC) C Compiler. Weakness variants typically do not heavy use of GCC extensions or attributes. However, there may be specific instances where GCC extensions were required to implement a target weakness. The following GCC versions were used to develop, test, and validate each weakness variant.

- GCC 4.4.7 (default provided by CentOS 6.5)
- GCC 4.6.3 (default provided by Ubuntu 12.04)

In addition to primarily testing the C weakness variants with GCC, each variant is also required to support Clang, as some proposed technologies leverage Clang for source code analysis and compilation. However, Clang was never the target compiler for any extensive testing.

Supported Java Runtime Environments

STONESOUP Test & Evaluation (T&E) targeted the Oracle Java 7 Runtime Environment. As such, all weakness variants were permitted to make use of any Java feature defined up through version 7. In most cases, the weakness variants remain Java 5 compliant in order to better support injection into various base programs.

Runtime Environment

Working Directory

Due to the fact that the developed weakness algorithmic variants are intended to be injected into arbitrary base programs, they expect in many cases a directory tree to exist within which input files will exist and output files will be generated. This workspace is standardized through the TEXAS execution system as:

```
/opt/stonesoup/workspace
```

Within this root workspace, the following directory are expected to exist.

```
/opt/stonesoup/workspace/testData
```

The weakness algorithmic variants expect to have full permissions to that directory, and they expect it to be their current working directory during execution. For some weakness algorithmic variants, this is not a requirement. However, it is strongly suggested that each variant be run in this manner to ensure proper execution.

Running Weaknesses

All weakness variants include a main() entry point to support debugging. The main() entry point takes exactly one argument, the taint source. This argument is then passed directly to a function/method, typically named "weakness", containing the weakness body.

The individual weakness variant snippets are not intended to be compiled as executables for the final Test & Evaluation (T&E) activity. Instead, each weakness will be injected into a larger base program. During this injection process, the weakness in pair with a dynamically generated taint source and one or more code complexity features may be injected between the taint source read and the start of the weakness. For more information on the injection process, please refer to the Test Case Generation documentation.

Supporting Scripts

Several of the weakness variants developed require one or more scripts to support the setup or triggering of the implemented vulnerability. In cases where these scripts are reused across multiple weakness variants, the source code has been included in Appendix B.

Concurrency Handling

This class contains weaknesses related to concurrently using shared resources. These weaknesses involve:

- multiple threads or processes using shared resources such as file or memory
- issues that arise from synchronization or locking

In general, concurrency issues are hard to identify, because they arise from the details of the schedule of multiple threads. Since the program itself does not have control of the schedule, the problems may arise only occasionally and seemingly randomly.

Most of the concurrency handling weaknesses rely on an external script to create the timing for the race condition that is being exploited. These weaknesses are complicated to debug, run, and test. It is our intention that concurrency weaknesses will fail almost all the time (>99%) on an exploit input, and succeed almost all the time (>99%) on a benign input. However, because we rely on the details of the schedule, and because these weaknesses by definition rely on race conditions, we cannot guarantee 100% accuracy.

Weakness Members

The concurrency handling weakness class is defined as a set of weakness types as defined by the MITRE Common Weakness Enumeration (CWE) ontology. The MITRE CWE ontology is defined as a tree, and as such may target CWEs that are a parent or child of another CWE. In those cases, CWEs may be proposed to remove from the test set target given that they are fully covered by one or more other CWEs. Individual weakness variants (i.e. tests) are designed around the definition, description, and code examples provided for each CWE.

CWE	Name	Target Source Languages
CWE-363	Race Condition Enabling Link Following	C, Java
CWE-367	Time-of-check Time-of-use (TOCTOU) Race Condition	C, Java
CWE-412	Unrestricted Externally Accessible Lock	C, Java
CWE-414	Missing Lock Check	C, Java
CWE-479	Signal Handler Use of a Non-reentrant Function	C
CWE-543	Use of Singleton Pattern Without Synchronization in a Multithreaded Context	C, Java
CWE-567	Unsynchronized Access to Shared Data in a Multithreaded Context	Java
CWE-572	Call to Thread run() instead of start()	Java
CWE-609	Double-Checked Locking	C, Java
CWE-663	Use of a Non-reentrant Function in a Concurrent Context	C, Java
CWE-764	Multiple Locks of a Critical Resource	C, Java
CWE-765	Multiple Unlocks of a Critical Resource	C, Java
CWE-820	Missing Synchronization	C, Java
CWE-821	Incorrect Synchronization	C, Java
CWE-828	Signal Handler with Functionality that is not Asynchronous-Safe	C
CWE-831	Signal Handler Function Associated with Multiple Signals	C
CWE-832	Unlock of a Resource that is not Locked	Java
CWE-833	Deadlock	C, Java

Weakness Variants

The following weakness variants are developed and for the C and Java source languages.

C Source Language

Error rendering macro 'detailssummary' : In template Confluence.Templates.Macro.MasterDetail.contentLink2: When evaluating "contextPath()": Error while computing function "contextPath()": null

Java Source Language

Error rendering macro 'detailssummary' : In template Confluence.Templates.Macro.MasterDetail.contentLink2: When evaluating "contextPath()": Error while computing function "contextPath()": null

Notes

The weakness variants developed to test the CWEs that define the weakness class generally fall into three categories.

Schedule-Independent Weaknesses

First, there are variants that do not require any influence of the runtime thread schedule to reliably reach the trigger point and cause the desired negative technical impact. Weakness that fall into this category can largely be tested an analyzed like any other non-concurrency related weakness.

For general information on the runtime environment, refer to Execution Environment.

Multi-Process Race Conditions

The second case, requiring simple timing delays, mostly applies to cases in which a single thread is accessing a resource, but the resource has changed through an external signal or environment change. Analyzing and running these weaknesses is discussed in Multi-Process Race Conditions.

Multi-Threaded Race Conditions

The third case, requiring a more robust external synchronization mechanism, includes cases where multiple threads are accessing the same internal resource, but the order in which those threads are scheduled can lead to undefined/unsafe behavior. Analyzing and running these weaknesses is discussed in Multi-Threaded Race Conditions.

Multi-Process Race Conditions

Several weaknesses require another process or the Operating System (OS) to take some action to exercise a race condition within the construct of a weakness variant. These types of actions may include, but are not limited to:

- Signaling the process representing the weakness variant
- Making a file system change after the weakness variant performs a "check"
- Consuming or locking another shared resource

In general, concurrency issues are hard to identify, because they arise from the details of the schedule of multiple processes. Since the program itself does not have control of the schedule, the problems may arise only occasionally and seemingly randomly.
Creating weaknesses that include race conditions is relatively straightforward. However, reliably running those weaknesses with inputs that result in a benign thread schedule or an exploiting thread schedule is difficult. For the purposes of STONESOUP Test & Evaluation (T&E), all weaknesses are required to have a set of good inputs that demonstrate "correct" or "normal" behavior, and another set of inputs that trigger the race condition. Due to this requirement, adding instructions, such as sleep(), are not possible because they would not support both execution schedules. Instead, an external lightweight synchronization process has been developed to allow for simple synchronization across multiple processes while enabling the code to remain free of any statements that may unfairly influence the schedule.

Approach

In the multi-process race conditions, there is typically only one thread executing in the weakness variant. The weakness performs some action that can be observed by an external process, and the external process reacts to it (for instance, by changing the file system or signaling the weakness). Meanwhile, the weakness delays long enough for the external process to take its action, and then continues its execution.

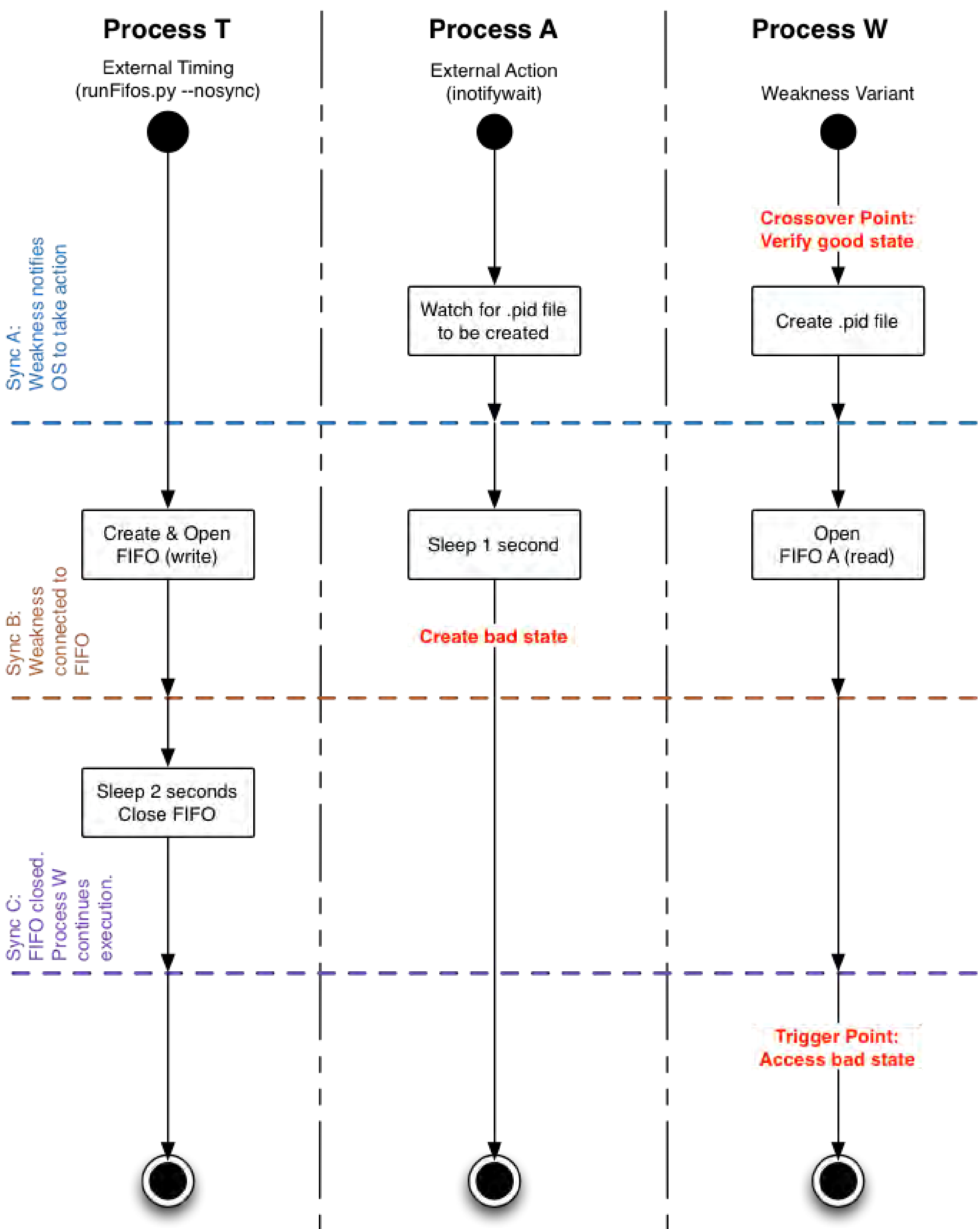
In these cases, we typically run two external processes.

1. External Action: inotifywait - a low-overhead process that monitors the file system, looking for a particular file to be created by the

weakness. When the file is created, the external process takes action that will subsequently affect the weakness (for instance, by altering the file system or signaling the weakness).
2. External Timing: runFifos.py --nosync - a process that uses FIFO files to create a delay in the weakness so that the External Action script can complete its task before the weakness continues. Opening a FIFO file blocks until there is a process available to read from other end. This blocking allows us to synchronize two separate processes at the FIFO open. The weakness then reads from the FIFO until the FIFO is closed, allowing us to synchronize again at the FIFO close.

The external synchronization strategy allows the weakness to have a true race condition in the implementation, but one that can be influenced to reliably observe benign and exploit behavior. For each weakness, all the exploit inputs use an External Timing script (runFifos.py --nosync) that creates FIFO files. The benign inputs do not use an External Timing script; instead they use regular files that are created before the weakness is run.

Bad Case Using FIFO File

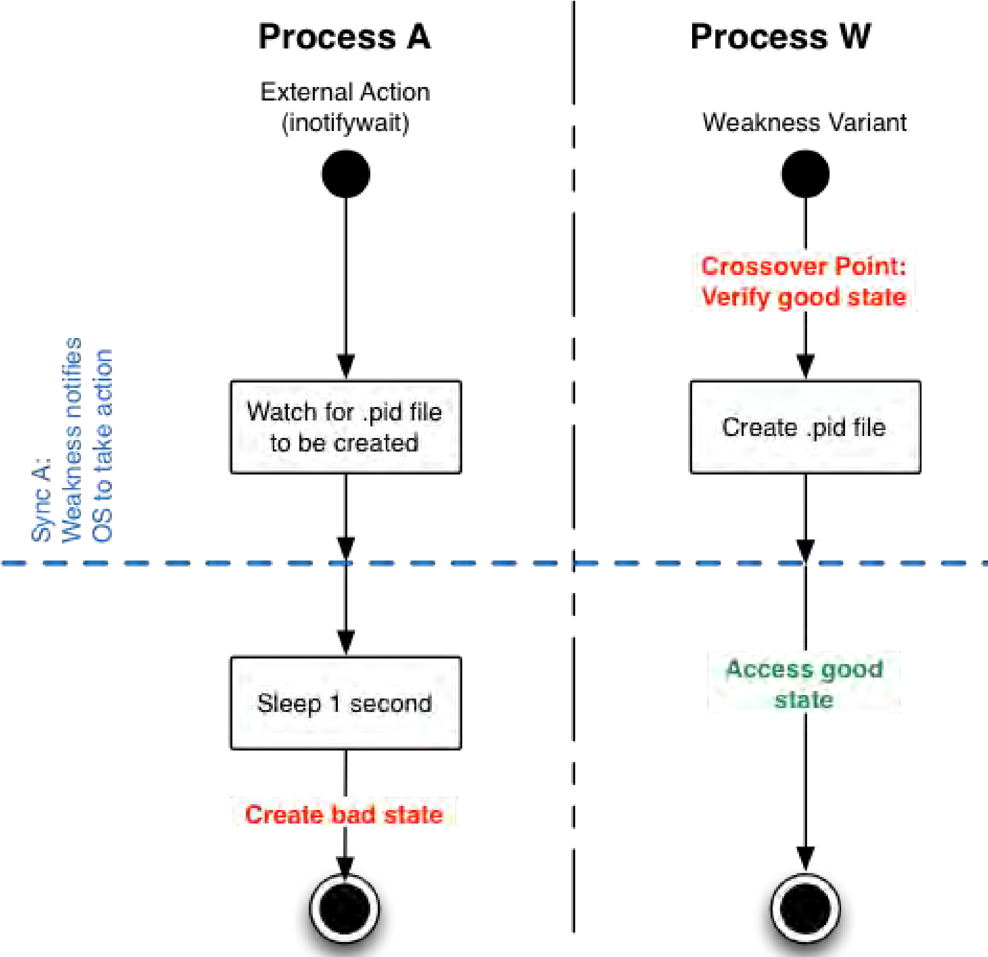


1. Process T (External Timing) begins, calling the runFifos.py script with the --nosync option. It opens a single FIFO for writing, which blocks until there is a process to read data on the other end. In our case, the weakness will read the data from the FIFO file.
2. Process A (External Action) begins, calling inotifywait. This script waits for a file with extension ".pid" to be created in a particular directory.
3. Process W (Weakness) begins, checks that some state is OK (for instance, that some file exists), and then creates a file called <input-file>.pid. When this step has occurred, we are at **Sync A**.

4. Process A (External Action) notices that <input-file>.pid has been created, sleeps for one second, and then alters the state to be bad (for instance, by deleting the file).
5. Process W (Weakness) starts reading from the FIFO, putting us at **Sync B**.
6. Process T (External Timing) sleeps for 2 seconds, then closes its FIFO, putting us at **Sync C**.
7. Process W (Weakness) accesses the bad state (for instance, by assuming that the file still exists), resulting in negative technical impact.

There is still a race condition here. The external scripts (Processes T and A) are controlling the timing. Process A (External Action) sleeps 1 second, then alters the state. Simultaneously, Process T (External Timing) sleeps 2 seconds, then accesses the state. If for some reason Process A (External Action) script does not run in the intervening 1 second, then the state may still be good when the weakness accesses it. However, the difference between a 1-second sleep and a 2-second sleep is significant, and we do not expect to hit this condition without performer intervention.

Good Case Using Regular File



In this case, we use a regular file that exists before the script begins, rather than a FIFO created by the runFifos.py script.

1. Process A (External Action) starts and waits for a file with extension ".pid" to be created in a particular directory.
2. Process W (Weakness Variant), checks that some state is OK (for instance, that some file exists), and then creates a file called <input-file>.pid. When this step has occurred, we are at **Sync A**.
3. Process W (Weakness Variant) reads a regular file which already exists, resulting in minimal delay.
4. Process W (Weakness Variant) accesses the good state, resulting in successful execution.
5. Meanwhile, Process A (External Action) notices that <input-file>.pid has been created, sleeps for one second, and then alters the state to be bad (for instance, by deleting the file).
   The 1-second sleep means that the change will almost certainly occur after Process W (Weakness Variant) has run to completion.

There is still a race condition here. After **Sync A**, Process W (Weakness Variant) runs, while Process A (External Action) sleeps 1 second, then alters the state. If for some reason Process W (Weakness Variant) does not run during the intervening 1 second, then the state may be bad when Process W (Weakness Variant) eventually does access the state. However, a 1-second sleep is long, and we do not expect to his this condition without performer intervention.

Runtime Information

Examine the YAML file in detail to determine what inputs are required.

For exploit inputs (all of which use FIFOs):

```
runFifos.py /opt/stonesoup/workspace/testData/fifo1 --nosync &
# Run inotify script from YAML file - details are different for different weaknesses
inotifywait ...
./weakness.out '/opt/stonesoup/workspace/testData/fifo1 <taint-input>'
killall runFifos.py inotifywait
rm /opt/stonesoup/workspace/testData/fifo1 /opt/stonesoup/workspace/testData/fifo2
```

For benign inputs (all of which use files):

```
touch /opt/stonesoup/workspace/testData/file1
# Note use of file 1 and file2, rather than fifo1 and fifo2 below
./weakness.out '/opt/stonesoup/workspace/testData/file1 <taint-input>'
```

Related Information

runFifos.py

Multi-Threaded Race Conditions

Many of the target weaknesses defined by this weakness class involve race conditions within a multi-threaded program. These race conditions typically manifest as two or more threads concurrently access a shared resource, such as variables or other memory segments. These race conditions differ from others defined in the weakness class because the resource in question is typically internal to the application, rather than a resource to which an external process may have access.
In general, concurrency issues are hard to identify, because they arise from the details of the schedule of multiple threads. Since the program itself does not have control of the schedule, the problems may arise only occasionally and seemingly randomly. In many cases, race conditions in a multi-threaded program can be avoided by implementing proper locking or using other lock-free strategies, such as atomic operations.

Creating weaknesses that include race conditions is relatively straightforward. However, reliably running those weaknesses with inputs that result in a benign thread schedule or an exploiting thread schedule is difficult. For the purposes of STONESOUP Test & Evaluation (T&E), all weaknesses are required to have a set of good inputs that demonstrate "correct" or "normal" behavior, and another set of inputs that trigger the race condition. Due to this requirement, adding instructions, such as sleep(), are not possible because they would not support both execution schedules. Instead, an external lightweight synchronization process has been developed to allow for simple synchronization across two threads while enabling the code to remain free of any statements that may unfairly influence the thread schedule.

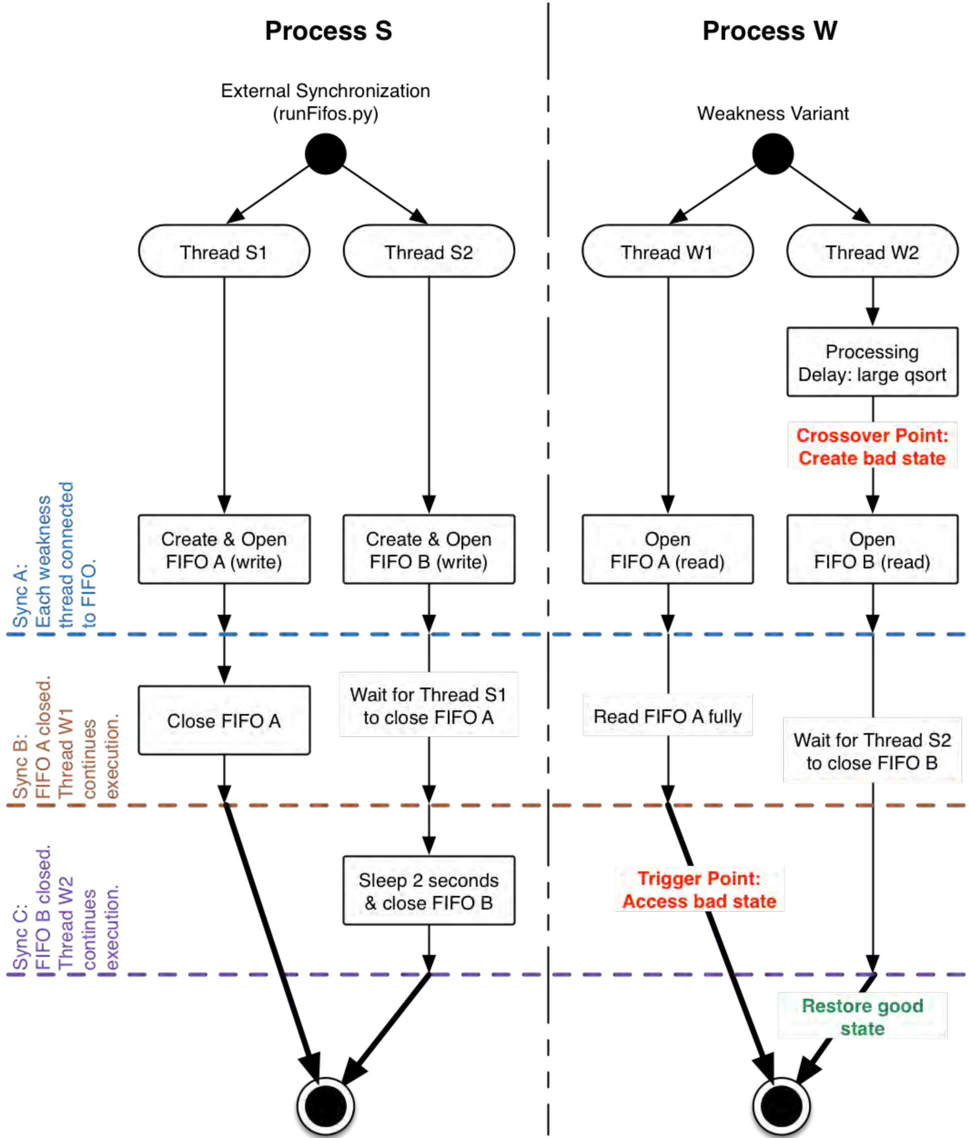
Approach

The multi-thread race conditions use the file system to synchronize and appropriately delay threads. At a high-level, an external process creates two FIFOs, one for each thread, and uses synchronization in the external process to wait for both threads to open and begin reading from the respective FIFOs. Once the external process has detect that both threads have reached the same read point, the external process lets one thread through first by finishing and closing one FIFO. Next, the external process sleeps for a short period of time and then finishes and closes the second FIFO. This blocking I/O strategy allows the kernel to appropriately context switch between the threads in the weakness variant, influencing the execution order.

One possible problem with the external synchronization strategy is the potential for deadlocking the weakness variant if a proposed technology decides to force the undesired thread schedule (i.e. chooses the incorrect execution order). To avoid this, the external synchronization process includes timeouts at the synchronization points to essentially give up and move forward if the other thread in the weakness variant has not reached the desired point in the weakness.

This external synchronization strategy allows the weakness to have a true race condition in the implementation, but one that can be influenced to reliably observe benign and exploit behavior. For each weakness, all the exploit inputs and most of the benign inputs use FIFO files created by the runFifos.py script. One good input uses regular files that are created before the weakness is run.

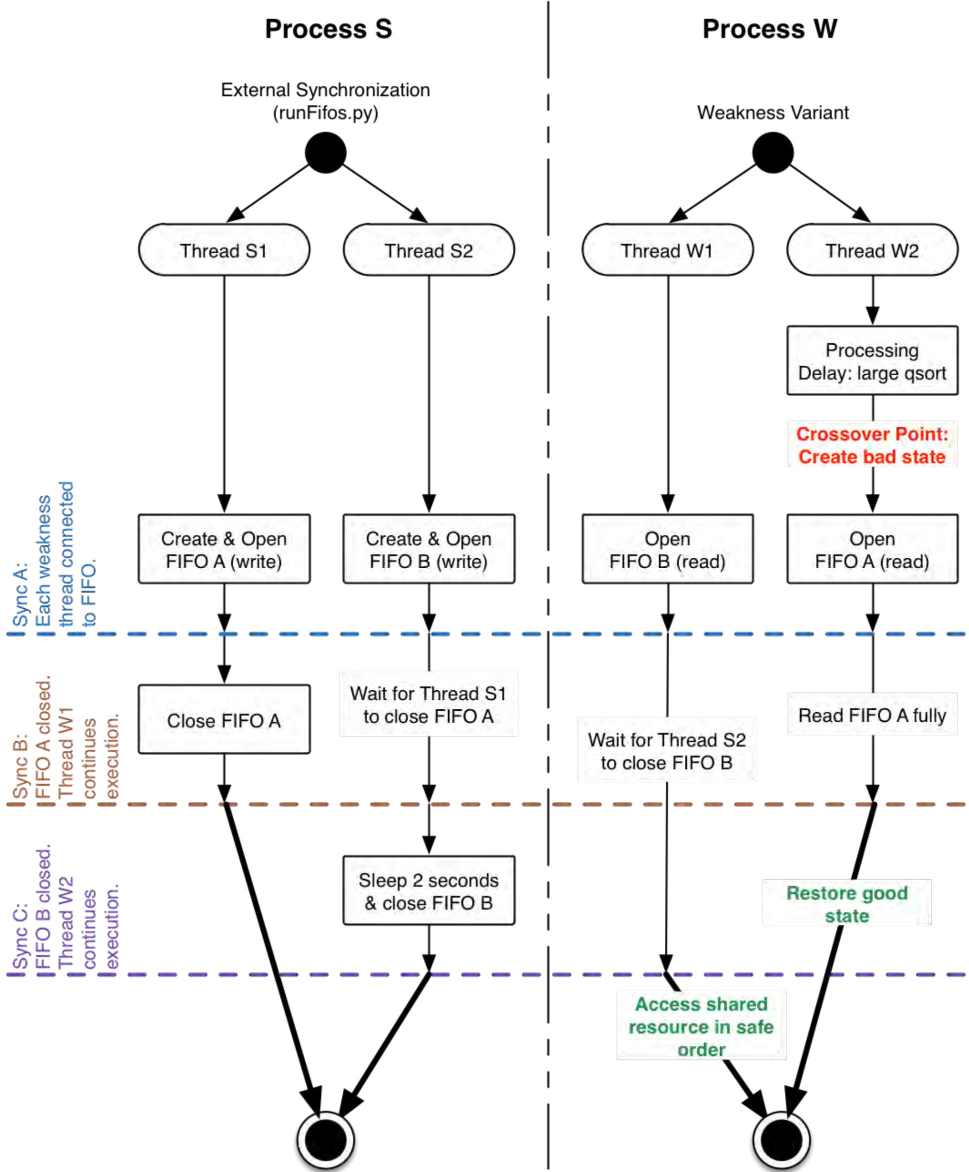
Exploit Case Using FIFO Files



1. The runFifos.py script begins and spawns two threads. Each thread opens a FIFO file for writing. The threads will not progress past this point until both threads have opened their FIFO file. Opening a FIFO file will block until there is a process to read the data on the other end. In our case, the weakness will read the data from these FIFO files. So these operations will block until each weakness thread has opened its FIFO file and started to receive data.
2. Thread W1 opens its FIFO file and starts to read data. Thread W2 runs a large quicksort (which exists only to create a delay in the regular-file case which we discuss later), sets some shared data to a bad state, and then opens its FIFO file and starts to read data. When these steps are done, we are at **Sync A**.
3. Thread S1 closes its FIFO file. When this step is done, we are at **Sync B**.
4. Thread W1 continues running, accesses the shared data (which is in a bad state), and causes a negative technical impact such as segmentation fault or deadlock.
5. At the same time, Thread S2 sleeps for 2 seconds and then closes its FIFO file. At the end of this action, we would be at **Sync C** if the negative technical impact had not occurred.
6. If the negative technical impact had not occurred, Thread W2 would set the shared data to a good value and the weakness would finish running.

There is still a race condition here. If Thread W1 does not get to run during the 2 seconds when Thread S2 is sleeping, then Thread W2 may set the shared data to a good state, causing Thread W1 will run without error. However, a 2-second sleep is very long, and we do not expect to hit this condition without intervention by the performer technology. In addition, we are forcing file I/O, which the run-time system knows is slow, and thus it is likely to cause a context switch.

Good Case Using FIFO Files



In the good case that uses FIFO files, we run the same script as for the bad case, except the weakness swaps which FIFO file each thread is using. In the bad case, Thread W1 uses FIFO 1 and Thread W2 uses FIFO 2. In the good case, Thread W1 uses FIFO 2 and Thread W2 uses FIFO 1.

Steps 1 -3 are identical in the good and bad cases.

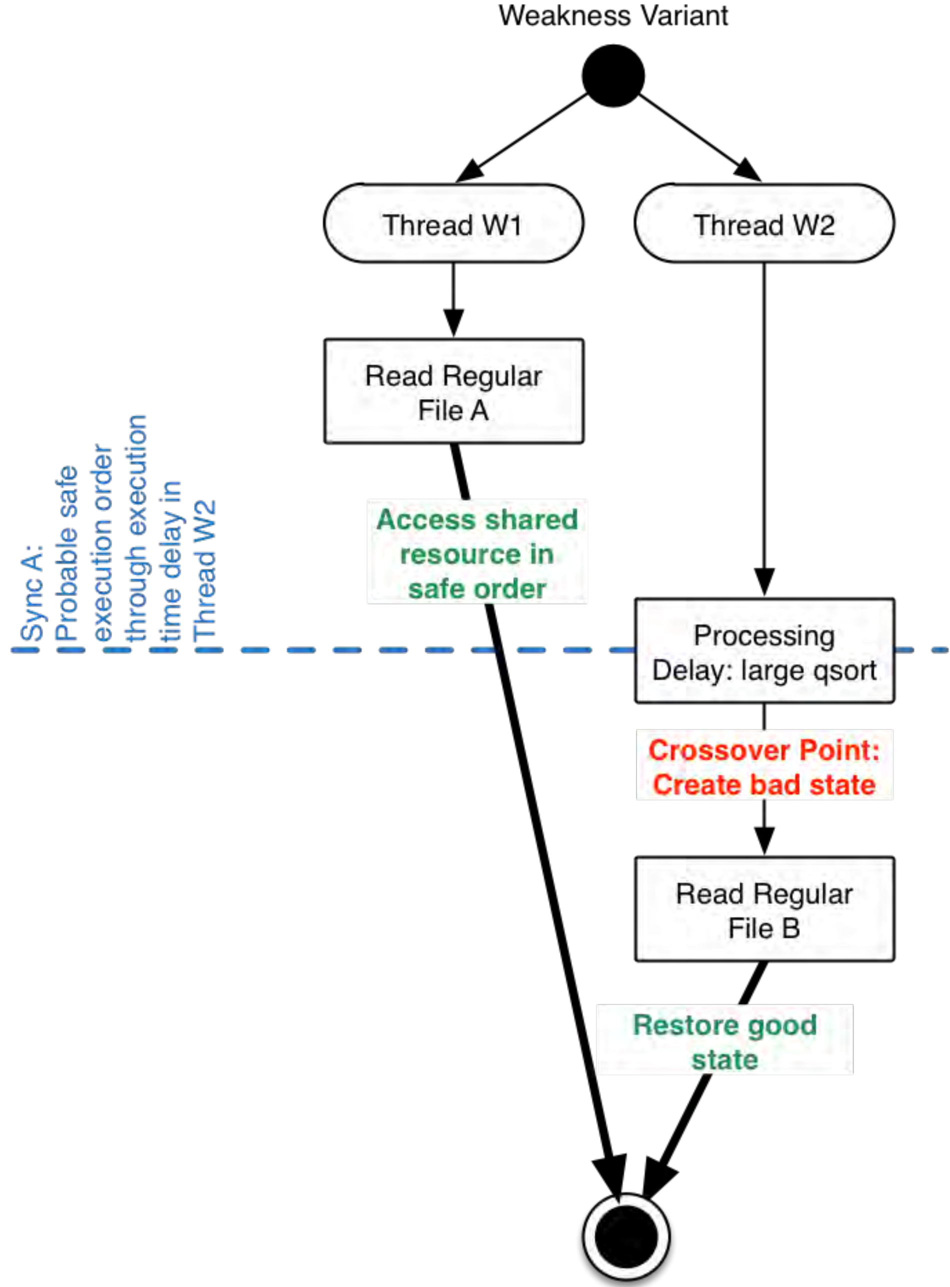
1. The runFifos.py script begins and spawns two threads. Each thread opens a FIFO file for writing. The threads will not progress past this point until both threads have opened their FIFO file. Opening a FIFO file will block until there is a process to read the data on the other end. In our case, the weakness will read the data from these FIFO files. So these operations will block until each weakness thread has opened its FIFO file and started to receive data.
2. Thread W1 opens its FIFO file and starts to read data. Thread W2 runs a large quicksort (which exists only to create a delay in the regular-file case which we discuss later), sets some shared data to a bad state, and then opens its FIFO file and starts to read data. When these steps are done, we are at **Sync A**.
3. Thread S1 closes its FIFO file. When this step is done, we are at **Sync B**.
4. Thread W2 runs, setting the shared data to a good state. (This is the first step that is different than the bad case.)
5. At the same time, Thread S2 sleeps for 2 seconds and then closes its FIFO file. We are now at **Sync C**.
6. Thread W1 runs, accessing the shared data (which is in a good state). The weakness runs successfully to completion.

There is still a race condition here: if Thread W2 does not get to run during the 2 seconds when Thread S2 is sleeping, then Thread W1 could access shared data in a bad state, resulting in a negative technical impact. However, a 2-second sleep is very long, and we do not expect to hit this condition without intervention by the performer technology. In addition, we are forcing file I/O, which the run-time system knows is slow, and thus it is likely to cause a context switch.

Approved for public release; distribution unlimited.

Good Case Using Regular Files

One benign input value for each weakness uses regular file that are created before the weakness is run, rather than FIFO files. Since the files already exist, there is no separate script process running beside the weakness. There are no synchronization points between the two threads in the weakness. We rely only on a large quick sort to delay one weakness thread and allow another to proceed.



1. Thread W2 does a large quick sort, while at the same time Thread W1 accesses shared data that is set to a good value. The quick sort exists only to delay the execution of Thread W2 and allow Thread W1 to proceed.
2. After Thread W2 finishes the quick sort, it sets the shared data to a bad state and then back to a good state.

There is still a race condition here, because we are not guaranteed that Thread W1 will access the shared data before the quick sort in Thread W2 completes. However, there is a very small window where the shared data is in a bad state, and it is unlikely that the other thread will try to access it in that time. We do not expect to hit this condition without intervention by the performer technology.

Runtime Information

Examine the YAML file in detail to determine what inputs are required.

For exploit inputs (all of which use FIFOs):

```
runFifos.py /opt/stonesoup/workspace/testData/fifo1
/opt/stonesoup/workspace/testData/fifo2 &
./weakness.out '<quicksort-size> /opt/stonesoup/workspace/testData/fifo1
/opt/stonesoup/workspace/testData/fifo2 <taint-input>'
killall runFifos.py
rm /opt/stonesoup/workspace/testData/fifo1 /opt/stonesoup/workspace/testData/fifo2
```

For benign inputs using FIFOs:

```
runFifos.py /opt/stonesoup/workspace/testData/fifo1
/opt/stonesoup/workspace/testData/fifo2 &
# Note that fifo2 and fifo1 are reversed below
./weakness.out '<quicksort-size> /opt/stonesoup/workspace/testData/fifo2
/opt/stonesoup/workspace/testData/fifo1 <taint-input>'
killall runFifos.py
rm /opt/stonesoup/workspace/testData/fifo1 /opt/stonesoup/workspace/testData/fifo2
```

For benign inputs using files:

```
touch /opt/stonesoup/workspace/testData/file1
/opt/stonesoup/workspace/testData/file2
# Note use of file 1 and file2, rather than fifo1 and fifo2 below
./weakness.out '<quicksort-size> /opt/stonesoup/workspace/testData/file1
/opt/stonesoup/workspace/testData/file2 <taint-input>'
```

Related Information

runFifos.py

C - CWE-363A - Race Condition Enabling Link Following

Summary

This snippet implements a time of check time of use vulnerability that allows arbitrary link following. The snippet takes in a control file and an input file. The input file is checked to see if it is in the current working directory and not a symbolic link. If both of these conditions are true, the snippet opens and reads the FIFO file and opens, reads, and prints the data in the input file. Since there is a delay between checking the validity of the file and opening it, the file can be replaced with a symbolic link, and the snippet can be tricked into following it. The control file is used as a method to delay execution long enough for this vulnerability to occur.

This is an example of a Multi-Process Race Condition.

This weakness differs from CWE-367 because CWE-367 is a general case time of check time of use weakness implemented as a DOS by

causing a nonexistent file to be read where as this weakness is specifically a time of check time of use vulnerability that allows data leakage through arbitrary link following.

Weakness Class	Concurrency Handling
CWE	CWE-363
Variant	A
Language	C
Status	WEAK-324 - CWE-363-0-C DONE
Negative Technical Impact(s)	Read File

Implementation

The snippet takes two string as input, the first is the name of a control file that is used for delaying execution and the second is the name of the file to be read. The snippet first checks whether the file name of the file to be read contains any '/' characters, ensuring that it is in the current directory.

```
int stonesoup_path_is_relative(char *path) {
    char *chr = 0;
    chr = strchr(path,'/');
    if (chr == 0) {
        stonesoup_printf("Path is relative\n");
        return 1;
    } else {
        stonesoup_printf("Path is not relative\n");
        return 0;
    }
}
```

If the file is in the current directory, it is also checked to make sure that it is not a symbolic link.

```
int stonesoup_isSymLink(char *file) {
    struct stat statbuf;

    if (lstat(file, &statbuf) < 0) {    /* if error occured */
        stonesoup_printf("Error accessing path.\n");
        return 1;                       /* just end program */
    }

    if (S_ISLNK(statbuf.st_mode) == 1) {
        stonesoup_printf("Path is symlink.\n");
        return 1;
    }
    stonesoup_printf("Path is valid.\n");
    return 0;
}
```

Assuming that the file is both in the current directory and not a symbolic link, the snippet proceeds to create a file called <FILE_NAME>.pid, explained below in the inotifywait section, and read the control file. This will cause a delay in execution, and possibly allow the time of check time of use vulnerability to be exploited. The call to this function, waitForChange(), is the crossover point.

```
void waitForChange(char* file, char* sleepFile) {
    int fd;
    char filename[500] = {0};

    stonesoup_printf("In waitForChange\n");

    strcat(filename, file);
    strcat(filename, ".pid");

    if ((fd = open(filename, O_CREAT|O_WRONLY, 0666)) == -1) {
        stonesoup_printf("Error opening file.");
    }
    else {
        if (write(fd, "q", sizeof(char)) == -1) {
            stonesoup_printf("Error writing to file.");
        }

        if (close(fd) == -1) {
            stonesoup_printf("Error closing file.");
        }
        stonesoup_readFile(sleepFile);
    }
}
```

If the control file delays the execution long enough for an external script to replace the input file, the snippet will open this unchecked file and print out the contents, possibly causing an information leak. This is the trigger point.

```
/* STONESOUP: TRIGGER-POINT (race condition enabling link following) */
fread(stonesoup_buffer,sizeof(char ),stonesoup_size,stonesoup_file);
stonesoup_buffer[stonesoup_size] = '\0';
stonesoup_printf(stonesoup_buffer);
fclose(stonesoup_file);
```

If the execution is not delayed long enough for an external script to replace the input file, the snippet will open the checked file and print the correct information, exiting without error.

External Scripts

The execution of this snippet in the bad cases is reliant on two external scripts which will be described in detail below.

runFifos.py

The runFifos.py script indirectly controls the time that the snippet takes between checking the validity of the input file and opening the file for reading. When run in the configuration required by this snippet, set by passing the script the --nosync flag, the script creates a FIFO file, writes 32 bytes of data to the file, and waits for it to be read. Once the first 32 bytes have been read, the script sleeps for 2 seconds and writes another 32 bytes to the file. After the second 32 bytes have been read the snippet closes the file and deletes it.

This script is able to control the amount of time between check and opening of the input file in the snippet due to the fact that FIFO files are blocking by default. The readFile call shown above will not return until the FIFO file is closed by the script, which has the file open for writing. This creates a robust way for influencing the execution time for the snippet.

The correct execution of this script for the snippets bad test cases is shown below.

```
runFifos.py /opt/stonesoup/workspace/testData/fifo1 --nosync
```

Where /opt/stonesoup/workspace/testData/fifo1 is the name of the control file that will be passed to the snippet when it is executed.

It is worth noting that the runFifos.py script must be executed as a backgrounded pre-process to the snippets execution, and allowed to continue during the execution of the snippet.

The source code for this script can be found here.

inotifywait

The inotifywait script is a one line bash script using the inotify-tools inotifywait command to check for the presence of a specific file created by the snippet, and upon finding that file, use it to identify the input file and replace it with a symbolic link. This script is essential to the proper functioning of the bad test cases.

The inotifywait script exists in the following form:

```
inotifywait -m -e create --format \"%f\" /opt/stonesoup/workspace/testData/ | while
read NEW_FILE; do if [[ \"${NEW_FILE}\" == *.pid ]]; then
REPLACE_FILE=\"${NEW_FILE%.pid}\"; sleep 1; echo \"Replacing: \"
\"/opt/stonesoup/workspace/testData/${REPLACE_FILE}\" && rm -f
\"/opt/stonesoup/workspace/testData/${REPLACE_FILE}\" && ln -s /etc/passwd
\"/opt/stonesoup/workspace/testData/${REPLACE_FILE}\" && echo \"Replaced.\"; else
echo \"Ignoring: \" \"/opt/stonesoup/workspace/testData/${NEW_FILE}\"; fi; done;
```

The script waits for the snippet to create a file in /opt/stonesoup/workspace/testData/ following the pattern <FILE_NAME>.pid where <FILE_NAME> is the name of the snippet's input file. Once the file has been detected the script sleeps for 1 second, deletes the snippets input file, and replaces it with a symbolic link to /etc/passwd.

The inotifywait script is run for each test case, good and bad, in order to more properly simulate the race condition present in the snippet. Because of this, the inotifywait utilizes a 1 second sleep before acting on the file in order to provide the snippet enough time to open and read the proper file in the good test cases, but still allow it to modify the file in the 2 second window allowed by the runFifos.py script in the bad test cases.

It is worth noting that the inotifywait script must be executed as a backgrounded pre-process to the snippets execution and allowed to continue during the execution of the snippet.

Benign Inputs

Benign inputs all follow the pattern of a short text file as the control file and a valid text file in the current working directory as the input file.

The use of a short text file for the control file ensures that the snippets execution time between checking the validity of the input file and reading the file is negligible in relation to the timing of the inotifywait script.

Examples are as follows:

```
/opt/stonesoup/workspace/testData/file1 goodfile1
```

Where `/opt/stonesoup/workspace/testData/file1` is a text file containing no text and `goodfile1` is a text file in the current working directory containing the text `"A non-empty file."`.

```
/opt/stonesoup/workspace/testData/file1 goodfile2
```

Where `/opt/stonesoup/workspace/testData/file2` is a text file containing no text and `goodfile2` is a text file in the current working directory containing the text `"This is a file."`.

```
/opt/stonesoup/workspace/testData/file1 goodfile3
```

Where `/opt/stonesoup/workspace/testData/file3` is a text file containing no text and `goodfile3` is a text file in the current working directory containing the text `"This is another file."`.

Exploiting Inputs

Exploiting inputs all follow the pattern of a FIFO file controlled by runFifos.py as the control file, and a valid text file in the current working directory as the input file.

The use of a FIFO file controlled by runFifos.py ensures that the snippets execution time between checking the validity of the input file and reading the file is significant enough that the exploit is triggered and the snippet is tricked into leaking data by following a symbolic link.

For the sake of clarity, we assume that the FIFO file in each test case has been created by running the runFifos.py script in the background as follows:

```
runFifos.py <FIFO_FILE_NAME> --nosync
```

Where `<FIFO_FILE_NAME>` is the path to the FIFO file used as a control file in the snippet, for example: `/opt/stonesoup/workspace/testData/fifo1`.

Examples are as follows:

```
/opt/stonesoup/workspace/testData/fifo1 badfile1
```

Where `/opt/stonesoup/workspace/testData/fifo1` is a FIFO file controlled by `runFifos.py` as shown above and `badfile1` is a text file containing the text `"This is yet another file."`.

```
/opt/stonesoup/workspace/testData/fifo1 badfile2
```

Where `/opt/stonesoup/workspace/testData/fifo1` is a FIFO file controlled by `runFifos.py` as shown above and `badfile2` is a text file containing the text `"This is also a file."`.

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Restrictions and Requirements

C - CWE-367A - Time-of-check Time-of-use (TOCTOU) Race Condition

Summary

This snippet implements a time of check time of use vulnerability that allows a DOS due to the input file being deleted before opening. The snippet takes in a control file and an input file. The input file is checked to see if it is in the current working directory and exists. If both of these conditions are true, the snippet opens and reads the control file and opens, reads, and prints the data in the input file. Since there is a delay between checking the validity of the file and opening it, the input file can be deleted before opening causing a DOS: Uncontrolled Exit. The control file is used as a method to delay execution long enough for this vulnerability to occur.

This is an example of a Multi-Process Race Condition .

This weakness differs from CWE-363 because CWE-363 is is specifically a time of check time of use vulnerability that allows data leakage through arbitrary link following, where as this weakness is a general case time of check time of use weakness implemented as a DOS by causing a nonexistent file to be read.

Weakness Class	Concurrency Handling
CWE	CWE-367
Variant	A
Language	C
Status	**WEAK-336** - CWE-367-0-C **DONE**
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

The snippet takes two string as input, the first is the name of a control file that is used for delaying execution and the second is the name of the file to be read. The snippet first checks whether the file name of the file to be read contains any '/' characters, ensuring that it is in the current directory.

```
int stonesoup_path_is_relative(char *path) {
    char *chr = 0;
    chr = strchr(path,'/');
    if (chr == 0) {
        stonesoup_printf("Path is relative\n");
        return 1;
    } else {
        stonesoup_printf("Path is not relative\n");
        return 0;
    }
}
```

If the file is in the current directory, it is also checked to make sure that the file exists.

```
int stonesoup_is_valid(char *path)
{
    if(access(path, F_OK) != -1) {
        stonesoup_printf("Path is accessible\n");
        return 1;
    }
    stonesoup_printf("Path is not accessible\n");

    return 0;
}
```

Assuming that the file is both in the current directory and exists, the snippet proceeds to create a file called <FILE_NAME>.pid, explained in the inotifywait section, and read the control file. This will cause a delay in execution, and possibly allow the time of check time of use vulnerability to be exploited. The call to this function is the crossover point.

```
void waitForChange(char* file, char* sleepFile) {
    int fd;
    char filename[500] = {0};

    stonesoup_printf("In waitForChange\n");

    strcat(filename, file);
    strcat(filename, ".pid");

    if ((fd = open(filename, O_CREAT|O_WRONLY, 0666)) == -1) {
        stonesoup_printf("Error opening file.");
    }
    else {
        if (write(fd, "q", sizeof(char)) == -1) {
            stonesoup_printf("Error writing to file.");
        }
        if (close(fd) == -1) {
            stonesoup_printf("Error closing file.");
        }
        stonesoup_readFile(sleepFile);
    }
}
```

If the control file delays the execution long enough for an external script to delete the input file, the snippet will attempt to open and read a nonexistent file, causing the snippet to crash. This is the trigger point.

```
/* STONESOUP: CROSSOVER-POINT (Time of Check, Time of Use) */
waitForChange(stonesoup_abs_path, stonesoup_sleep_file);
/* STONESOUP: TRIGGER-POINT (Time of Check, Time of Use) */
stonesoup_file = fopen(stonesoup_abs_path,"rb");
fseek(stonesoup_file,0,2);
{...}
```

If the execution is not delayed long enough for an external script to delete the input file, the snippet will open the input file correctly and echo the contents to stdOut, exiting without error.

External Scripts

The execution of this snippet in the bad cases is reliant on two external scripts which will be described in detail below.

runFifos.py

The runFifos.py script indirectly controls the time that the snippet takes between checking the validity of the input file and opening the file for reading. When run in the configuration required by this snippet, set by passing the script the --nosync flag, the script creates a FIFO file, writes 32 bytes of data to the file, and waits for it to be read. Once the first 32 bytes have been read, the script sleeps for 2 seconds and writes another 32 bytes to the file. After the second 32 bytes have been read the snippet closes the file and deletes it.

This script is able to control the amount of time between check and opening of the input file in the snippet due to the fact that FIFO files are blocking by default. The readFile call shown above will not return until the FIFO file is closed by the script, which has the file open for writing. This creates a robust way for influencing the execution time for the snippet.

The correct execution of this script for the snippets bad test cases is shown below.

```
runFifos.py /opt/stonesoup/workspace/testData/fifo1 --nosync
```

Where `/opt/stonesoup/workspace/testData/fifo1` is the name of the control file that will be passed to the snippet when it is executed.

It is worth noting that the runFifos.py script must be executed as a backgrounded pre-process to the snippets execution, and allowed to continue during the execution of the snippet.

The source code for this script can be found here.

inotifywait

The inotifywait script is a one line bash script using the inotify-tools inotifywait command to check for the presence of a specific file created by the snippet, and upon finding that file, use it to identify the input file and replace it with a symbolic link. This script is essential to the proper functioning of the bad test cases.

The inotifywait script exists in the following form:

```
inotifywait -m -e create --format \"%f\" /opt/stonesoup/workspace/testData/ | while
read NEW_FILE; do if [[ \"${NEW_FILE}\" == *.pid ]]; then
REPLACE_FILE=\"${NEW_FILE%.pid}\"; sleep 1; echo \"Deleting: \"
\"/opt/stonesoup/workspace/testData/${REPLACE_FILE}\" && rm -f
\"/opt/stonesoup/workspace/testData/${REPLACE_FILE}\" && echo \"Deleted.\"; else
echo \"Ignoring: \" \"/opt/stonesoup/workspace/testData/${NEW_FILE}\"; fi; done;
```

The script waits for the snippet to create a file in `/opt/stonesoup/workspace/testData/` following the pattern `<FILE_NAME>.pid` where `<FILE_NAME>` is the name of the snippet's input file. Once the file has been detected the script sleeps for 1 second and deletes the snippets input file.

The inotifywait script is run for each test case, good and bad, in order to more properly simulate the race condition present in the snippet. Because of this, the inotifywait utilizes a 1 second sleep before acting on the file in order to provide the snippet enough time to open and read the proper file in the good test cases, but still allow it to modify the file in the 2 second window allowed by the runFifos.py script in the bad test cases.

It is worth noting that the inotifywait script must be executed as a backgrounded pre-process to the snippets execution and allowed to continue during the execution of the snippet.

Benign Inputs

Benign inputs all follow the pattern of a short text file as the control file and a valid text file in the current working directory as the input file.

The use of a short text file for the control file ensures that the snippets execution time between checking the validity of the input file and reading the file is negligible in relation to the timing of the inotifywait script.

Examples are as follows:

```
/opt/stonesoup/workspace/testData/file1 goodfile1
```

Where `/opt/stonesoup/workspace/testData/file1` is a text file containing no text and `goodfile1` is a text file in the current working directory containing the text `"A non-empty file."`.

```
/opt/stonesoup/workspace/testData/file2 goodfile2
```

Where `/opt/stonesoup/workspace/testData/file2` is a text file containing no text and `goodfile2` is a text file in the current working directory containing the text `"This is a file."`.

```
/opt/stonesoup/workspace/testData/file3 goodfile3
```

Where `/opt/stonesoup/workspace/testData/file3` is a text file containing no text and `goodfile3` is a text file in the current working directory containing the text `"This is another file."`.

Exploiting Inputs

Exploiting inputs all follow the pattern of a FIFO file controlled by runFifos.py as the control file, and a valid text file in the current working directory as the input file.

The use of a FIFO file controlled by runFifos.py ensures that the snippets execution time between checking the validity of the input file and reading the file is significant enough that the exploit is triggered and the snippet crashes by attempting to open and read a nonexistent file.

For the sake of clarity, we assume that the FIFO file in each test case has been created by running the runFifos.py script in the background as follows:

```
runFifos.py <FIFO_FILE_NAME> --nosync
```

Where `<FIFO_FILE_NAME>` is the path to the FIFO file used as a control file in the snippet, for example: `/opt/stonesoup/workspace/testData/fifo1`.

Examples are as follows:

```
/opt/stonesoup/workspace/testData/fifo1 badfile1
```

Where `/opt/stonesoup/workspace/testData/fifo1` is a FIFO file controlled by `runFifos.py` as shown above and `badfile1` is a text file containing the text `"This is yet another file."`.

```
/opt/stonesoup/workspace/testData/fifo2 badfile2
```

Where `/opt/stonesoup/workspace/testData/fifo2` is a FIFO file controlled by `runFifos.py` as shown above and `bad_0.txt` is a text file containing the text `"This is also a file."`.

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Restrictions and Requirements

C - CWE-412A - Unrestricted Externally Accessible Lock

Summary

This snippet takes the name of an externally accessible file as input, and treats the file as a global mutex lock. The snippet will attempt to "grab" the mutex lock by checking for the files existence, and creating it if it doesn't exist. The creation of the file is treated as grabbing the lock, and if the snippet is able to do so it will enter the critical section and return, deleting the file to "release" the lock, and finishing execution without error. However, if the file already exists, the snippet will continually poll the existence of the file, waiting to be able to "grab" the lock by creating the file, resulting in a paused state for as long as the file exists.

Weakness Class	Concurrency Handling
CWE	CWE-412
Variant	A
Language	C
Status	WEAK-338 - CWE-412-0-C DONE
Negative Technical Impact(s)	DOS: Infinite Loop

Implementation

This snippet takes in the name of an externally accessible file as input and creates a full path to the file, appending the ".lck" file extension to the end. This is the crossover point.

```
int stonesoup_fd;
char *stonesoup_extension = ".lck";
char *stonesoup_thelockfile;
stonesoup_thelockfile =  (char *)malloc(strlen(STONESOUP_TESTDATA) +
strlen(STONESOUP_TAINT_SOURCE) +
                              strlen(stonesoup_extension) + 1);
if (stonesoup_thelockfile) {
    /* STONESOUP: CROSSOVER-POINT (Deadlock) */
    strcpy(stonesoup_thelockfile, STONESOUP_TESTDATA);
    strcat(stonesoup_thelockfile, STONESOUP_TAINT_SOURCE);
    strcat(stonesoup_thelockfile, stonesoup_extension);
```

The snippet then attempts create the file if it doesn't exists, looping until it is able to do so. If the file already exists, this will result in a paused state for as long as the file exists. This is the trigger point.

```
/* STONESOUP: TRIGGER-POINT (Deadlock) */
while ((stonesoup_fd = open(stonesoup_thelockfile, O_CREAT | O_EXCL, 0644) == -1) )
{
    sleep(0);
}
```

If the file does not exists, the snippet will create the file, print to the output stream that it has grabbed the lock, and finally delete the file again, completing execution without error.

Benign Inputs

All valid names of nonexistent files that are writable by the snippet are benign.

All of the benign inputs create 3 files before execution: `/opt/stonesoup/workspace/testData/file1.lck`, `/opt/stonesoup/workspace/testData/sync/file2.lck`, and `/opt/stonesoup/workspace/testData/temp/sync/file3.lck`

Examples of benign inputs are as files:

```
critical
```

Were `critical` does not exist.

```
sync/noissues
```

Where `sync/noissues` does not exist.

```
temp/sync/section
```

Where `temp/sync/section` does not exist.

Exploiting Inputs

All valid names of currently existing files that are writable by the snippet are exploitative.

All of the exploiting inputs create 3 files before execution: `/opt/stonesoup/workspace/testData/file1.lck`, `/opt/stonesoup/workspace/testData/sync/file2.lck`, and `/opt/stonesoup/workspace/testData/temp/sync/file3.lck`

Examples of exploiting inputs are as files:

```
file1
```

```
sync/file2
```

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Restrictions and Requirements

C - CWE-414A - Missing Lock Check

Summary

This snippet implements a missing lock check that allows two threads to access a shared character array simultaneously, leading to a null pointer deference. It takes an integer, the names of two control files, and an input string as input. The integer and two control files are used for timing, and are discussed in more detail below. The snippet creates two threads that both modify a shared character array containing the input string, however only one of the the threads implements a locking mechanism. Since one thread sets the character array pointer to null temporarily and the other tries to read the character array, this can cause a null pointer dereference if the timing occurs properly.

This is an example of a Multi-Threaded Race Condition.

This snippet differs from the snippet for CWE-820 because the snippet for CWE-820 spawns two threads and neither thread implements a locking mechanism while both access a shared object.

This snippet differs from the snippet for CWE-821 because the snippet for CWE-821 spawns two threads which each use a separate lock object.

Weakness Class	Concurrency Handling
CWE	CWE-414
Variant	A
Language	C
Status	WEAK-342 - CWE-414-0-C DONE
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

This snippet contains two main functions: toCap and delNonAlpha, each executed by a separate thread. Both functions act upon a shared struct (constructed from the input), however, only toCap implements a mutex lock before accessing the shared memory. The threads executing both functions are spawned at the same time, and execute concurrently. The execution of each function is explained below.

delNonAlpha

The delNonAlpha thread is called with a pointer to a shared struct containing the snippet's input values. Once executing, this thread creates a temporary character array and iterates through the shared character array, copying all alpha characters into the new array. Once the temp array contains all of the alpha characters from the shared array, the snippet enters the crossover point by setting the pointer to the shared array to NULL. The control file is then read, allowing our test case to influence the execution timing of the thread (explained in the next section).

```
char* temp = malloc(sizeof(char) * (stonesoupData->data_size + 1));
while(stonesoupData->data[i] != '\0') {
    if((stonesoupData->data[i] >= 'A' && stonesoupData->data[i] <= 'Z') ||
       (stonesoupData->data[i] >= 'a' && stonesoupData->data[i] <= 'z')) {
        temp[j++] = stonesoupData->data[i];
    }
    i++;
}
temp[j++] = '\0';
stonesoupData->data_size = j;
free(stonesoupData->data);
/* STONESOUP: CROSSOVER-POINT (missinglockcheck) */
stonesoupData->data = NULL;                                 /* sets global ptr to
null, ka-boom */
stonesoup_readFile(stonesoupData->file2);
```

Once the control file is finished reading, assuming the bad case has not been executed causing a null pointer dereference (explained below), the thread sets the shared string equal to the newly filled temp string.

toCap

The toCap thread is called with a pointer to a shared struct containing the snippet's input values. Once executing, this thread locks a mutex lock, and reads its control file (used for delaying the snippets execution, explained below). The thread then allocates an integer array with size equal to the input integer, and populates the newly created array with integers in descending order (from size to 0). This array is then sorted to ascending order. The purpose of the sort is to slow down the execution of the snippet in certain good test cases (explained in more detail in the next section).

Once the array sort is finished, the thread enters the trigger point by accessing the shared character array and converting it to upper case. If the other thread has set the shared character array pointer to NULL but not yet reassigned it a value the array access will result in a null pointer dereference.

```
pthread_mutex_lock(&stonesoup_mutex);
stonesoup_readFile(stonesoupData->file1);
stonesoup_arr = malloc(sizeof(int) * stonesoupData->qsize);
for (stonesoup_i = 0; stonesoup_i < stonesoupData->qsize; stonesoup_i++) {
    stonesoup_arr[stonesoup_i] = stonesoupData->qsize - stonesoup_i;
}
qsort(stonesoup_arr, stonesoupData->qsize, sizeof(int), &stonesoup_comp);
free(stonesoup_arr);
for(stonesoup_i = 0; stonesoup_i < stonesoupData->data_size; stonesoup_i++) {
    /* STONESOUP: TRIGGER-POINT (missinglockcheck) */
    if (stonesoupData->data[stonesoup_i] >= 'a' &&
        stonesoupData->data[stonesoup_i] <= 'z') {           /* null pointer
dereference possible */
        stonesoupData->data[stonesoup_i] -= 32;
    }
}
pthread_mutex_unlock(&stonesoup_mutex);
```

If the other snippet has finished execution and reset the shared character array pointer to a valid character array, the string access will occur normally, the thread will unlock it's mutex, and the snippet will finish execution without error.

Timing Explanation

Due to the random nature of race conditions, we cannot rely on the thread scheduler alone to run our threads and expect that they will deterministically result in either a good or bad outcome. In order to provide a deterministic execution environment we introduced two control mechanisms into the threads themselves. The first is a set of two control files which, when created as FIFO files and controlled by the runFifos.py script described below, ensure that one file read completes two seconds before the other. The other control method is to pass in two regular files for the control files, essentially negating the readFile() commands, and instead passing a significantly large integer in order to influence the thread timing though an array sort on an array of descending integers with size equal to the input integer.

For more information on either of these approaches, view the documentation for multi-threaded race conditions found here.

runFifos.py

The runFifos.py script indirectly controls the time execution order of the threads in a snippet, the specific configuration described below is set by passing two file names of nonexistent files to the runFifos.py script. The script will create both files and control them to ensure that the first FIFO closes two seconds before the second, allowing them to be used to control the execution of the snippet. This is done by writing 32 bytes of data to both FIFOs and waiting for them both to be read, ensuring that both threads in the snippet are synchronized on the file read. The script then writes a second 32 bytes to the first FIFO file and closes it. It then waits two seconds and writes a second 32 bytes to the second FIFO file and closes it as well. The order is as follows:

```
t1: wrote first 32 bytes to file
t2: wrote first 32 bytes to file
ALL: threads synchronized
t1: wrote second 32 bytes to file
t2: t1 finished
t2: wrote second 32 bytes to file
```

The correct execution of this script for the snippets bad test cases is shown below.

```
runFifos.py /opt/stonesoup/workspace/testData/fifo1
/opt/stonesoup/workspace/testData/fifo2
```

Where `/opt/stonesoup/workspace/testData/fifo1` and `/opt/stonesoup/workspace/testData/fifo2` are the names of the control files that will be passed to the snippet when it is executed.

It is worth noting that the runFifos.py script must be executed as a backgrounded pre-process to the snippets execution and allowed to continue to run during the execution of the snippet.

The source code for this script can be found here.

Benign Inputs

Benign inputs come in two forms. The first is using FIFO files with the order passed to the snippet reversed from the order called in the `runFifos.py` script. When executing benign test cases in this format the value of the input integer does not change the execution of the snippet and only serves to slow the overall execution of the snippet when the value is sufficiently large. The second format is to call the snippet with two regular files as control files and a significantly large control integer.

For the following examples that use FIFO files we will assume that the `runFifos.py` script is executed in the format show above.

FIFO Files

Examples of benign inputs with FIFO control files are as follows:

```
10 /opt/stonesoup/workspace/testData/fifo2 /opt/stonesoup/workspace/testData/fifo1
hephalump
```

```
1 /opt/stonesoup/workspace/testData/fifo2 /opt/stonesoup/workspace/testData/fifo1
Sniffles_and_whiffles
```

Regular Files

Examples of benign inputs with regular control files are as follows:

```
5000000 /opt/stonesoup/workspace/testData/file1
/opt/stonesoup/workspace/testData/file2 sIlly_rabbit,_tricks_are_for_kids.
```

Exploiting Inputs

Exploring inputs all use FIFO control files passed in the same order as they are called in the `runFifos.py` script. Like the benign test case using FIFO control files above, the value of the integer input is irrelevant and simply serves to slow the overall execution of the snippet when the value is sufficiently large. For example:

```
1 /opt/stonesoup/workspace/testData/fifo1 /opt/stonesoup/workspace/testData/fifo2
May_the_force_be_with_you.
```

```
72 /opt/stonesoup/workspace/testData/fifo1 /opt/stonesoup/workspace/testData/fifo2
What_is_the_answer_to_life,_the_universe,_and_everything?
```

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Restrictions and Requirements

C - CWE-479A - Signal Handler Use of a Non-reentrant Function

Summary

This snippet implements a non-reentrant function that is called by a signal handler. The snippet takes a control file and input string as input. The control file is used for timing and the input string is used as data for the snippet to manipulate. The snippet assigns a signal handler that calls a non-reentrant function that locks a mutex lock and manipulates the input data. If the snippet is currently executing the non-reentrant function when the signal handler is called, it will re-enter the function and deadlock on the call to lock the mutex.

This is an example of a Multi-Process Race Condition.

Weakness Class	Concurrency Handling
CWE	CWE-479
Variant	A
Language	C
Status	**WEAK-344** - CWE-479-0-C **DONE**
Negative Technical Impact(s)	DOS: Deadlock

Implementation

This snippet takes two strings as input. The first string is the name of a control file used for timing, and the second string is an arbitrary input string used as data for the snippet to operate on. The snippet starts by initializing a mutex lock and assigning a signal handler, and then calls the delNonAlpha() function.

```
if (pthread_mutex_init(&stonesoup_mutex, NULL) != 0) {
    stonesoup_printf("Mutex failed to initilize.");
}
if (signal(SIGUSR1, sig_handler) == SIG_ERR) {       /* setup signal handler */
    stonesoup_printf ("Error catching SIGUSR1!\n");
}
delNonAlpha(stonesoupData);
```

The assigned signal handler also calls delNonAlpha() which is the crossover point.

```
void sig_handler (int sig) {
    /* STONESOUP: CROSSOVER-POINT (nonreentrentsighandler) */
    if (stonesoupData != NULL) {
        delNonAlpha(stonesoupData);         /* call non-reentrant function -
deadlock */
    }

    signal(SIGUSR1, SIG_IGN);
}
```

Once the snippet is executing delNonAlpha(), it locks the mutex, and allocates a new character array. This locking of the mutex is the trigger point when this function is called from the signal handler. The snippet then iterates through the input string copying all alpha characters into the newly allocated string. The input string is then free()'d and it's pointer is set to point to the new string. The snippet then calls and enters waitForSig().

```
/* STONESOUP: TRIGGER-POINT (nonreentrant signal handler) */
pthread_mutex_lock(&stonesoup_mutex);       /* mutex lock causes deadlock on
re-entrance */
while(stonesoupData->data[i] != '\0') {
    if((stonesoupData->data[i] >= 'A' && stonesoupData->data[i] <= 'Z') ||
       (stonesoupData->data[i] >= 'a' && stonesoupData->data[i] <= 'z')) {
        temp[j++] = stonesoupData->data[i];
    }
    i++;
}
temp[j++] = '\0';
stonesoupData->data_size = j;
free(stonesoupData->data);
stonesoupData->data = temp;
waitForSig(stonesoupData->file1);       /* Deadlock */
```

Once in waitForSig() the snippet creates a file with the name "<PID>.pid" where <PID> is the snippets current process ID. Once the file is created, the snippet opens and starts reading the control file, pausing for two seconds if the snippet was called with a FIFO file controlled by runFifos.py (explained below), or returning immediately if called with a regular file for a control file.

```
void waitForSig(char* sleepFile) {
    int fd;
    char outStr[25] = {0};
    char filename[500] = {0};



    stonesoup_printf("In waitForSig\n");

    sprintf(outStr, "%d.pid", getpid());
    strcat(filename, STONESOUP_TESTDATA);
    strcat(filename, outStr);

    if ((fd = open(filename, O_CREAT|O_WRONLY, 0666)) == -1) {
        stonesoup_printf("Error opening file.");
    }
    else {
        if (write(fd, "q", sizeof(char)) == -1) {
            stonesoup_printf("Error writing to file.");
        }


        if (close(fd) == -1) {
            stonesoup_printf("Error closing file.");
        }
        stonesoup_readFile(sleepFile);
    }
}
```

If the snippet pauses execution in waitForSig(), an external script (inotifywait, explained below) will signal the snippet with a SIGUSR1 which will invoke the signal handler. The signal handler will then call the delNonAlpha() function again, deadlocking on the mutex lock. If the snippet does not pause, it will return from waitForSig() and finish execution without error, "deregistering" the signal handler at the end of execution by ignoring all future SIGUSR1 signals.

External Scripts

The execution of this snippet in the bad cases is reliant on two external scripts which will be described in detail below.

runFifos.py

The runFifos.py script indirectly controls the time that the snippet takes between checking the validity of the input file and opening the file for reading. When run in the configuration required by this snippet, set by passing the script the --nosync flag, the script creates a FIFO file, writes 32 bytes of data to the file, and waits for it to be read. Once the first 32 bytes have been read, the script sleeps for 2 seconds and writes another 32 bytes to the file. After the second 32 bytes have been read the snippet closes the file and deletes it.

This script is able to control the amount of time between check and opening of the input file in the snippet due to the fact that FIFO files are blocking by default. The readFile call shown above will not return until the FIFO file is closed by the script, which has the file open for writing. This creates a robust way for influencing the execution time for the snippet.

The correct execution of this script for the snippets bad test cases is shown below.

```
runFifos.py /opt/stonesoup/workspace/testData/fifo1 --nosync
```

Where /opt/stonesoup/workspace/testData/fifo1 is the name of the control file that will be passed to the snippet when it is executed.

It is worth noting that the runFifos.py script must be executed as a backgrounded pre-process to the snippets execution, and allowed to continue during the execution of the snippet.

The source code for this script can be found here.

inotifywait

The inotifywait script is a one line bash script using the inotify-tools inotifywait command to check for the presence of a specific file created by the snippet, and upon finding that file, use it to identify the input file and replace it with a symbolic link. This script is essential to the proper functioning of the bad test cases.

The inotifywait script exists in the following form:

```
inotifywait -m -e create --format \"%f\" /opt/stonesoup/workspace/testData/ | while
read NEW_FILE; do if [[ \"${NEW_FILE}\" == *.pid ]]; then
PROC_ID=\"${NEW_FILE%.pid}\"; echo \"Signaling: \" \"${PROC_ID}\" && echo
\"Waiting\" && sleep 0.5 && kill -s 10 \"${PROC_ID}\" && echo \"Signaled.\"; else
echo \"Ignoring: \" \"/opt/stonesoup/workspace/testData/${NEW_FILE}\"; fi; done;
```

The script waits for the snippet to create a file in /opt/stonesoup/workspace/testData/ following the pattern <PID>.pid where <PID> is the snippet's process ID. Once this file has been detected the script sleeps for 0.5 seconds and sends a SIGURS1 signal to the process ID indicated by the file.

The inotifywait script is run for each test case, good and bad, in order to more properly simulate the race condition present in the snippet. Because of this, the inotifywait utilizes a 0.5 second sleep before signaling the PID in order to provide the snippet enough time to finish exeuction in the good test cases, but still allow it to exploit the weakness in the 2 second window allowed by the runFifos.py script in the bad test cases.

It is worth noting that the inotifywait script must be executed as a backgrounded pre-process to the snippets execution and allowed to continue during the execution of the snippet.

Benign Inputs

Benign inputs all follow the pattern of a short text file as the control file and a string as input.

The use of a short text file for the control file ensures that the snippet completes execution before the inotifywait script is able to send it the SIGUSR1 signal.

Examples are as follows:

```
/opt/stonesoup/workspace/testData/file1 asdf
```

```
/opt/stonesoup/workspace/testData/file1 fD1Sa886
```

```
/opt/stonesoup/workspace/testData/file1 all5alpha5numeric5yo.
```

Exploiting Inputs

Exploiting inputs all follow the pattern of a FIFO file controlled by runFifos.py as the control file and an input string.

The use of a FIFO file controlled by runFifos.py ensures that the snippets execution time during the call to waitForSig() is significant enough that the exploit is triggered and the snippet is put into a deadlock state.

For the sake of clarity, we assume that the FIFO file in each test case has been created by running the runFifos.py script in the background as follows:

```
runFifos.py <FIFO_FILE_NAME> --nosync
```

Where `<FIFO_FILE_NAME>` is the path to the FIFO file used as a control file in the snippet, for example: `/opt/stonesoup/workspace/testData/fifo1`.

Examples are as follows:

```
/opt/stonesoup/workspace/testData/fifo1 :(){:|:&};:
```

Where `/opt/stonesoup/workspace/testData/fifo1` is a FIFO file controlled by `runFifos.py` as shown above and `badfile1` is a text file containing the text `"This is yet another file."`.

```
/opt/stonesoup/workspace/testData/fifo1 re-entrant,_re-smentrant
```

Where `/opt/stonesoup/workspace/testData/fifo1` is a FIFO file controlled by `runFifos.py` as shown above and `badfile2` is a text file containing the text `"This is also a file."`.

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Restrictions and Requirements

C - CWE-543A - Use of Singleton Pattern Without Synchronization in a Multithreaded Context

Summary

This snippet implements a singleton struct without synchronization that can lead to two threads receiving separate instances of the singleton struct resulting in a deadlocked state. It takes a control integer, the names of two control files, and another integer as input. The control integer and the two control files are used for timing within the snippet to ensure that it hits either a good or bad case, and the input integer is used as the number of integers to pass through the queue. The snippet then spawns two threads, one to push the numbers 0 to the input integer to a queue, and one to read the values from the queue and print them to the output stream. The shared queue is implemented as a singleton and is a blocking queue, if both threads create a separate instance of the singleton they will block indefinitely causing a deadlock.

This is an example of a Multi-Threaded Race Condition.

Weakness Class	Concurrency Handling
CWE	CWE-543
Variant	A
Language	C

Status	WEAK-409 - CWE-543-0-C DONE
Negative Technical Impact(s)	DOS: Deadlock

Implementation

This snippet contains three main functions: `stonesoup_calc_data()`, `stonesoup_print_data()`, and `get_instance()`, the first two are run concurrently by two threads, and the third is a singleton function used to return an instance of a queue used to share data between threads. Both threaded functions make a call to `get_instance()`. Assuming that the singleton performed as expected, one thread pushes a sequence of numbers from 0 to n to the shared queue, where n is the input integer, and the other grabs the data from the shared queue and prints it to the output stream.

stonesoup_calc_data()

This thread is called with a pointer to a shared struct containing the snippet's input values. Once executing, it allocates an integer array with size equal to the input integer, and populates the newly created array with integers in descending order (from size to 0). This array is then sorted to ascending order. The purpose of the sort is to slow down the execution of the snippet in certain good test cases (explained in more detail in the next section). The thread then reads its control file (used for delaying the snippets execution, explained below), and calls `get_instance()`. When the function call returns, the thread iterates from 0 to the input integer and pushes each number to the queue returned by `get_instance()`. If the call to `get_instance()` returned a unique instance of the shared queue, the enqueue call will block if enough data is pushed to it to fill it's buffer since it will not be read from by the other thread. This blocking will cause the `stonesoup_calc_data()` thread to deadlock. This is the second trigger point.

```
qsort_arr = malloc(sizeof(int)*ssD->qsize);
    if (qsort_arr != NULL) {
        for (i = 0; i < ssD->qsize; i++) {
            qsort_arr[i] = ssD->qsize - i;
        }
        qsort(qsort_arr, ssD->qsize, sizeof(int), &stonesoup_comp);
        free (qsort_arr);
        qsort_arr = NULL;
    }
stonesoup_readFile(ssD->file1);
ssQ = get_instance(NULL);
for (i = 0; i < ssD->data; i++) {
    /* STONESOUP2: TRIGGER-POINT (singletonpatternwithoutsync) */
    if (enqueue(ssQ, i) == -1) {
        break;
    }
}
enqueue(ssQ, -1);
```

If the call to `get_instance()` returns a shared queue like it is intended to do, this thread will write all of it's data to the queue, write a closing '-1' to the queue, and exit without error.

stonesoup_print_data()

This thread is called with a pointer to a shared struct containing the snippet's input values. Once executing, the thread calls `get_instance()`, and uses the returned queue to read data from and print that data to the stonesoup output stream. If the returned queue is unique, the thread will block indefinitely waiting for data to read, resulting in a deadlock. This is the trigger point.

```
struct stonesoup_queue *ssQ = get_instance(ssD->file2);
int i;

/* STONESOUP: TRIGGER-POINT (singletonpatternwithoutsync) */
while ((i = dequeue(ssQ)) != -1) {
    stonesoup_printf("Data: %d\n", i);
}
```

If the queue returned from the function call is shared with the other thread as intended this thread will read data from the queue and print it to the stonesoup output stream, exiting without error when the other thread ends it's data stream.

get_instance()

The get_instance function is a singleton function used to control access to a shared queue. When called, the function checks to see if the static instance of the queue is initialized and if not allows the current caller to enter the body of the function where the queue get's initialized. The issue arrises when two thread call the function concurrently, if both thread enter the body of the function each will end up with a separate reference of the queue when they return. In order to guarantee that this race condition occurs in a bad test case one of the threads (the first to call this function) passes a control file to this function, and reads this control file once the original null check passes. This allows the thread to pause execution before allocating a queue giving the other thread time to enter and allocate its own instance of the queue before the first thread continues. This is the crossover point.

```
/* STONESOUP: CROSSOVER-POINT (singletonpatternwithoutsync) */
if (ssQ == NULL) {
    if (file2 != NULL) {
        stonesoup_readFile(file2);
    }
    ssQ = (struct stonesoup_queue *)calloc(1, sizeof(struct stonesoup_queue));
    pthread_mutex_init(&(ssQ->lock), NULL);
    pthread_cond_init(&(ssQ->is_empty), NULL);
    pthread_cond_init(&(ssQ->is_full), NULL);
    ssQ->size = 0;
    ssQ->capacity = 30;
    ssQ->head = NULL;
    ssQ->tail = NULL;
}

return ssQ;
```

In a good test case the control files are passed in such a way that one thread is guaranteed to return from this function before the other thread calls it, causing both threads to return the same instance of the queue and the snippet to function properly.

Timing Explanation

Due to the random nature of race conditions, we cannot rely on the thread scheduler alone to run our threads and expect that they will deterministically result in either a good or bad outcome. In order to provide a deterministic execution environment we introduced two control mechanisms into the threads themselves. The first is a set of two control files which, when created as FIFO files and controlled by the runFifos.py script described below, ensure that one file read completes two seconds before the other, the order of which is described in the next section. The other control method is to pass in two regular files for the control files, essentially negating the readFile() commands, and instead passing a significantly large integer in order to influence the thread timing though an array sort on an array of descending integers with size equal to the input integer.

For more information on either of these approaches, view the documentation for multi-threaded race conditions found here, and read the section above about the JobHandler class which goes into more detail about how these control mechanisms work in this particular snippet.

runFifos.py

The runFifos.py script indirectly controls the time execution order of the threads in a snippet, the specific configuration described below is set by passing two file names of nonexistent files to the runFifos.py script. The script will create both files, and control them to ensure that the first

FIFO closes two seconds before the second, allowing them to be used to control the execution of the snippet. This is done by writing 32 bytes of data to both FIFOs and waiting for them both to be read, ensuring that both threads in the snippet are synchronized on the file read. The script then writes a second 32 bytes to the first FIFO file, and closes it. It then waits two seconds and writes a second 32 bytes to the second FIFO file and closes it as well. The order is as follows:

```
t1: wrote first 32 bytes to file
t2: wrote first 32 bytes to file
ALL: threads synchronized
t1: wrote second 32 bytes to file
t2: t1 finished
t2: wrote second 32 bytes to file
```

The correct execution of this script for the snippets bad test cases is shown below.

```
runFifos.py /opt/stonesoup/workspace/testData/fifo1
/opt/stonesoup/workspace/testData/fifo2
```

Where `/opt/stonesoup/workspace/testData/fifo1` and `/opt/stonesoup/workspace/testData/fifo2` are the names of the control files that will be passed to the snippet when it is executed.

It is worth noting that the runFifos.py script must be executed as a backgrounded pre-process to the snippets execution, and allowed to continue during the execution of the snippet.

The source code for this script can be found here.

Benign Inputs

Benign inputs come in two forms. The first is using FIFO files with the order passed to the snippet reversed from the order called in the `runFifos.py` script. When executing benign test cases in this format the value of the control integer does not change the execution of the snippet, and only serves to slow the overall execution of the snippet when the value is sufficiently large. The second format is to call the snippet with two regular files as control files and a significantly large control integer, used to control the execution of the threads.

For the following examples that use FIFO files we will assume that the `runFifos.py` script is executed in the format show above.

FIFO Files

Examples of benign inputs with FIFO control files are as follows:

```
0 /opt/stonesoup/workspace/testData/fifo2 /opt/stonesoup/workspace/testData/fifo1 50
```

```
12 /opt/stonesoup/workspace/testData/fifo2 /opt/stonesoup/workspace/testData/fifo1
13
```

Regular Files

Examples of benign inputs with regular control files are as follows:

```
1254312 /opt/stonesoup/workspace/testData/file1
/opt/stonesoup/workspace/testData/file2 61
```

Exploiting Inputs

Exploring inputs all use FIFO control files passed in the same order as they are called in the `runFifos.py` script. Like the benign test case using FIFO control files above, the value of the control input is irrelevant and simply serves to slow the overall execution of the snippet when the value is sufficiently large. For example:

```
15 /opt/stonesoup/workspace/testData/fifo1 /opt/stonesoup/workspace/testData/fifo2
66
```

```
74 /opt/stonesoup/workspace/testData/fifo1 /opt/stonesoup/workspace/testData/fifo2
51
```

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Restrictions and Requirements

C - CWE-609A - Double-Checked Locking

Summary

This snippet implements a double checked lock around the initialization of a shared struct in an attempt to be efficient. The snippet takes a control integer, the names of two control files, and an input string. The control integer and the two control files are used for timing within the snippet to ensure that we hit either a good or bad case, and the input string is used as shared data for the threads to act upon. When executing, the snippet spawns two threads which in turn attempt to get a shared instance of the input string by calling a double-checked locking function that initializes and/or returns a shared instance of a struct containing the input string. If one thread calls this function while the other one is inside the function initializing the struct it can return an uninitialized copy of the instance to the second thread. This will lead to an access of uninitialized data resulting in a StringIndexOutOfBoundsException.

This is an example of a Multi-Threaded Race Condition.

Weakness Class	Concurrency Handling
CWE	CWE-609
Variant	A
Language	C
Status	**WEAK-360** - CWE-609-0-C **DONE**
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

This snippet contains three main functions: `doStuff()`, `doStuff2()`, and `init_stonesoup_data2()`. The first two functions are run concurrently by two separate threads, and the third function is used to implement the weakness and contains a double checked lock. The three functions are described below.

doStuff()

The doStuff thread is called with a pointer to a shared struct containing the snippet's input values. Once executing, the thread calls `init_stonesoup_data2()` and returns.

```
ssD2 = init_stonesoup_data2((struct stonesoup_data*)ssD);
```

doStuff2()

The doStuff2 thread is called with a pointer to a shared struct containing the snippet's input values. Once executing, the thread allocates an integer array with size equal to the control integer, and populates the newly created array with integers in descending order (from size to 0). This array is then sorted to ascending order. The purpose of the sort is to slow down the execution of the snippet in certain good test cases (explained in more detail in the next section). Once the array sort is finished, the thread reads it's control file (again to control the execution time of the thread, explained below), and calls `init_stonesoup_data2()`.

```
stonesoup_arr = malloc(sizeof(int) * ssD->qsize);
for (stonesoup_i = 0; stonesoup_i < ssD->qsize; stonesoup_i++) {
    stonesoup_arr[stonesoup_i] = ssD->qsize - stonesoup_i;
}
qsort(stonesoup_arr, ssD->qsize, sizeof(int), &stonesoup_comp);
free(stonesoup_arr);
stonesoup_readFile(ssD->file1);
ssD2 = init_stonesoup_data2((struct stonesoup_data*)ssD);
```

Once the call to `init_stonesoup_data2()` returns, the snippet checks the first character of the data string contained in the struct that the function call returned. If the call to `init_stonesoup_data2()` returned before the data struct was fully initialized than this will result in a null pointer dereference. This is the trigger point.

```
/* STONESOUP: TRIGGER-POINT (double-check locking) */
if (ssD2->data2[0] != '\0') {
    stonesoup_printf("%s\n", ssD2->data2);
}
```

If the data is initialized when the call to `init_stonesoup_data2()` returns the thread will finish execution without error.

init_stonesoup_data2()

The `init_stonesoup_data2()` function checks to see if the ssD2 struct pointer is set to null and if not, it locks a mutex lock and checks to see if ssD2 struct pointer is still null. If the pointer is still null the function will allocate space for a new struct. The function then reads a control file in order to pause the execution temporarily. This is the crossover point, and allows one thread to create an instance of the struct and pause before initializing it. At this point, if another thread were to call the `init_stonesoup_data2()` function, it would check to see if ssD2 is null, and since it already points to an uninitialized chunk of memory, return without initializing the data. If the data is then accessed after the thread returns, a null pointer exception will occur, causing the snippet to segfault.

```
/* STONESOUP: CROSSOVER-POINT (double-check locking) */
if (ssD2 == NULL) {
    pthread_mutex_lock(&stonesoup_mutex);
    if (ssD2 == NULL) {
        stonesoup_printf("Initializing\n");
        ssD2 = calloc(1, sizeof(struct stonesoup_data2));
        stonesoup_readFile(ssD->file2);
        ssD2->data       = ssD->data;
        ssD2->qsize      = ssD->qsize;
        ssD2->data_size  = ssD->data_size;
        ssD2->data2      = ssD->data;
        ssD2->data_size2 = ssD->data_size;
    } else {
        stonesoup_printf("No need to initialize\n");
    }
    pthread_mutex_unlock(&stonesoup_mutex);
} else {
    stonesoup_printf("Data is already initialized\n");
}
return ssD2;
```

If the first thread finishes executing this function before the second thread calls it, the data will be properly initialized for both threads upon returning and the snippet will execute without error.

Timing Explanation

Due to the random nature of race conditions, we cannot rely on the thread scheduler alone to run our threads and expect that they will deterministically result in either a good or bad outcome. In order to provide a deterministic execution environment we introduced two control mechanisms into the threads themselves. The first is a set of two control files which, when created as FIFO files and controlled by the runFifos.py script described below, ensure that one file read completes two seconds before the other. The other control method is to pass in two regular files for the control files, essentially negating the readFile() commands, and instead passing a significantly large integer in order to influence the thread timing though an array sort on an array of descending integers with size equal to the input integer.

For more information on either of these approaches, view the documentation for multi-threaded race conditions found here.

runFifos.py

The runFifos.py script indirectly controls the time execution order of the threads in a snippet, the specific configuration described below is set by passing two file names of nonexistent files to the runFifos.py script. The script will create both files and control them to ensure that the first FIFO closes two seconds before the second, allowing them to be used to control the execution of the snippet. This is done by writing 32 bytes of data to both FIFOs and waiting for them both to be read, ensuring that both threads in the snippet are synchronized on the file read. The script then writes a second 32 bytes to the first FIFO file and closes it. It then waits two seconds and writes a second 32 bytes to the second FIFO file and closes it as well. The order is as follows:

```
t1: wrote first 32 bytes to file
t2: wrote first 32 bytes to file
ALL: threads synchronized
t1: wrote second 32 bytes to file
t2: t1 finished
t2: wrote second 32 bytes to file
```

The correct execution of this script for the snippets bad test cases is shown below.

```
runFifos.py /opt/stonesoup/workspace/testData/fifo1
/opt/stonesoup/workspace/testData/fifo2
```

Approved for public release; distribution unlimited.

Where `/opt/stonesoup/workspace/testData/fifo1` and `/opt/stonesoup/workspace/testData/fifo2` are the names of the control files that will be passed to the snippet when it is executed.

It is worth noting that the runFifos.py script must be executed as a backgrounded pre-process to the snippets execution and allowed to continue to run during the execution of the snippet.

The source code for this script can be found here.

Benign Inputs

Benign inputs come in two forms. The first is using FIFO files with the order passed to the snippet reversed from the order called in the `runFifos.py` script. When executing benign test cases in this format the value of the control integer does not change the execution of the snippet and only serves to slow the overall execution of the snippet when the value is sufficiently large. The second format is to call the snippet with two regular files as control files and a significantly large control integer.

For the following examples that use FIFO files we will assume that the `runFifos.py` script is executed in the format show above.

FIFO Files

Examples of benign inputs with FIFO control files are as follows:

```
5 /opt/stonesoup/workspace/testData/fifo2 /opt/stonesoup/workspace/testData/fifo1
a1s2d3f4
```

```
25 /opt/stonesoup/workspace/testData/fifo2 /opt/stonesoup/workspace/testData/fifo1
waffles
```

Regular Files

Examples of benign inputs with regular control files are as follows:

```
5000000 /opt/stonesoup/workspace/testData/file1
/opt/stonesoup/workspace/testData/file2 bannana.
```

Exploiting Inputs

Exploring inputs all use FIFO control files passed in the same order as they are called in the `runFifos.py` script. Like the benign test case using FIFO control files above, the value of the control input is irrelevant and simply serves to slow the overall execution of the snippet when the value is sufficiently large. For example:

```
50 /opt/stonesoup/workspace/testData/fifo1 /opt/stonesoup/workspace/testData/fifo2
This_will_surely_break_things.
```

```
75 /opt/stonesoup/workspace/testData/fifo1 /opt/stonesoup/workspace/testData/fifo2
asdfasdfasdfasdfasdfasdfasdfasdfasdf
```

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Restrictions and Requirements

C - CWE-663A - Use of a Non-reentrant Function in a Concurrent Context

Summary

This snippet implements a non-reentrant function that uses a static integer to iterate through a string setting each character to null. The snippet takes a control integer and an input string. The control integer is used for timing within the snippet to ensure that we hit either a good or bad case, and the input string is used as shared data for the threads to act upon. When executing, the snippet spawns two threads which both in turn call a non-reentrant function that iterates through the input string using a static integer as a counter. If both threads enter the function within a significantly small time frame the static integer will get incremented twice for each position in the string overflowing the array and overwriting the free() struct causing a segfault when the memory is free()'d.

This is an example of a variant on the Multi-Threaded Race Condition that only uses sorting for timing.

Weakness Class	Concurrency Handling
CWE	CWE-663
Variant	A
Language	C
Status	WEAK-362 - CWE-663-0-C DONE
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

This snippet contains three main function: `replaceSymbols()`, `toCaps()`, and `arrFunc()`. The first two functions are run concurrently by two separate threads and the third function is used to implement the trigger point by iterating through the shared string and overwriting characters in a non-reentrant manner. The three functions are described below.

replaceSymbols()

The replaceSymbols thread is called with a pointer to a shared struct containing the snippet's input values. Once executing, it grabs the mutex lock, and iterates through the shared input string replacing any symbols with '_', and releases the mutex lock. Once the thread has finished replacing the symbols in the input string it makes a call to `arrFunc()`. If the weakness occurs (explained below) then the call to `arrFunc()` will not return before the snippet segfaults.

```
pthread_mutex_lock(&stonesoup_mutex);
for(i = 0; i < stonesoupData->data_size; i++) {
    if (((stonesoupData->data[i] >= '!' && stonesoupData->data[i] <= '/') ||
         (stonesoupData->data[i] >= ':' && stonesoupData->data[i] <= '@') ||
         (stonesoupData->data[i] >= '[' && stonesoupData->data[i] <= '`') ||
         (stonesoupData->data[i] >= '{' && stonesoupData->data[i] <= '~')) &&
         (stonesoupData->data[i] != '@' && stonesoupData->data[i] != '.'))
    {
        stonesoupData->data[i] = '_';
    }
}
pthread_mutex_unlock(&stonesoup_mutex);
arrFunc(stonesoupData);
```

Assuming that the weakness has not occurred, the thread will return from the function call and join back with the main thread without error.

toCaps()

The toCaps thread is called with a pointer to a shared struct containing the snippet's input values. Once executing, it allocates an integer array with a size of 500000, and populates the newly created array with integers in descending order (from size to 0). This array is then sorted to ascending order. The purpose of the sort is to slow down the execution of the snippet to allow good test cases to succeed (explained in more detail in the next section). The thread then grabs the mutex lock, converts all of the alpha characters in the input string to capitals, and releases the mutex lock. Once the thread has finished converting the input string to capitals it makes a call to `arrFunc()`. If the weakness occurs (explained below) then the call to `arrFunc()` will not return before the snippet segfaults.

```
/* slow things down to make correct thing happen in good cases */
stonesoup_arr = malloc(sizeof(int)*threadTiming);
if (stonesoup_arr != NULL) {
    for (stonesoup_j = 0; stonesoup_j < threadTiming; stonesoup_j++) {
        stonesoup_arr[stonesoup_j] = threadTiming - stonesoup_j;
    }
    qsort(stonesoup_arr, threadTiming, sizeof(int), &stonesoup_comp);
    free (stonesoup_arr);
    stonesoup_arr = NULL;
}
pthread_mutex_lock(&stonesoup_mutex);
for(i = 0; i < stonesoupData->data_size; i++) {
    if(stonesoupData->data[i] >= 'a' && stonesoupData->data[i] <= 'z') {
        stonesoupData->data[i] -= 'a' - 'A';
    }
}
pthread_mutex_unlock(&stonesoup_mutex);
arrFunc(stonesoupData);
```

Assuming that the weakness has not occurred, the thread will return from the function call and join back with the main thread without error.

arrFunc()

The `arrFunc()` function is called with a pointer to the shared struct containing the snippet's input data. Once the function has been called, it enters a for loop that iterates through the shared input string, incrementing a global static integer 'stonesoup_ctr' that is used to index the string, replacing each character with the null character. This indexing of the string is the crossover point. If two threads are in this function simultaneously 'stonesoup_ctr' will get incremented twice per iteration of the for loop causing it to overflow the string, overwriting the struct that malloc() places in memory. When the string is free()'d by the main thread, the struct containing information about what memory to release will be corrupt causing a segfault.

```
for(stonesoup_i = 0;                                              /* and count
twice when second thread is in func */
    stonesoup_i < stonesoupData->data_size;                       /* which
will blow the free() struct away */
    stonesoup_i++, stonesoup_ctr++)
{
    /* STONESOUP: CROSSOVER-POINT (nonreentrant function in multi-threaded context)
*/
    stonesoupData->data[stonesoup_ctr] = '\0';
    stonesoup_printf("I: %d, C: %d\n", stonesoup_i, stonesoup_ctr);
    if (stonesoupData->qsize > 5) {
        fp = fopen("asdfqwer1234", "w+");
        fprintf(fp, "%d", stonesoup_i);
        fclose(fp);
        for (stonesoup_j = 0; stonesoup_j < stonesoupData->qsize; stonesoup_j++) {
            stonesoup_arr[stonesoup_j] = stonesoupData->qsize - stonesoup_j;
        }
        qsort(stonesoup_arr, stonesoupData->qsize, sizeof(int), &stonesoup_comp);
    }
}
```

Each iteration of the loop is slowed down by an array sort (like the one found in `toCaps()`) followed by a file access if the control integer is 5 or greater. When executing a bad test case the array sort causes the currently executing thread to slow its execution allowing the other thread time to execute and the file read is to ensure that a context switch occurs if the two threads are not operating it true concurrency (for instance, if the threads were being run on a single cpu core).

If two threads are not executing this function simultaneously, the function will perform as expected and an exception will not get thrown.

Timing Explanation

Due to the random nature of race conditions, we cannot rely on the thread scheduler alone to run our threads and expect that they will deterministically result in either a good or bad outcome. In order to provide a deterministic execution environment we introduce a control mechanism into the threads themselves. This control mechanism is a series if array sorts that are used to slow down certain threads in order to allow the other thread(s) time to execute while the these thread are in a certain state.

For more information on timing approaches, view the documentation for multi-threaded race conditions found here.

Benign Inputs

Benign inputs all use a control integer that is less than 5 to allow the threadTiming in replaceSymbols to slow down that thread for long enough that toCaps can return from `arrFunc()` before replaceSymbols enters it and to ensure that the file read in `arrFunc()` does not occur, thus not forcing a context switch. For example:

```
3 hephalump
```

```
1 Sniffles_and_whiffles,_yo.
```

```
5 Silly_rabbit,_tricks_are_for_kids!
```

Exploiting Inputs

Benign inputs all use a significantly large control integer in order to allow time for both threads to enter arrFunc() at relatively the same time, causing the weakness to occur. For example:

```
5000000 I'm_a_poet_and_I_didn't_even_know_it.
```

```
6000000 But_my_toes_showed_it._Theyre_long_fellows
```

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Restrictions and Requirements

C - CWE-764A - Multiple Locks of a Critical Resource

Summary

This snippet implements two threads that lock a counting semaphore multiple times, causing a deadlock if the lock is locked more times that it is unlocked. The snippet takes a string as input and if the input string contains spaces spawns two threads, the first of which locks a semaphore twice which causes this thread, and more than likely the second thread to deadlock on their calls to lock the semaphore. If the input string does not contain spaces only the second thread is spawned and the snippet completes without error.

This is an example of a Multi-Threaded Race Condition.

Weakness Class	Concurrency Handling
CWE	CWE-764
Variant	A
Language	C
Status	**WEAK-366** - CWE-764-0-C **DONE**
Negative Technical Impact(s)	DOS: Deadlock

Implementation

This snippet contains two main functions: `toCap()` and `replaceSpace()`. The snippet is called with a single input string, and if this input string contains spaces both functions are run concurrently, and if the input string does not contain spaces only `toCap()` is run by a single thread. The first method is the bad case and the second is the good. The snippet implements a counting semaphore instantiated to a value of 1 and uses this as a tool for implementing the weakness.

```
if (stonesoup_hasSpaces == 1) {
    /* STONESOUP: CROSSOVER-POINT (multiple locks) */
    if (pthread_create(&stonesoup_t0, NULL, replaceSpace, NULL) != 0) {
        stonesoup_printf("Thread 0 failed to spawn.");
    }
}
if (pthread_create(&stonesoup_t1, NULL, toCap, NULL) != 0) {
    stonesoup_printf("Thread 1 failed to spawn.");
}
```

The two threads are described below:

toCap()

This thread is called without any parameters, and once executing simply decrements the counting semaphore, iterates through the input string converting it to capitals, increments the counting semaphore, and exits.

```
int stonesoup_i = 0;
stonesoup_printf("Capitalizing input\n");
sem_wait(&stonesoup_sem);
while(stonesoup_global_str[stonesoup_i] != '\0') {
    if (stonesoup_global_str[stonesoup_i] > 'a' && stonesoup_global_str[stonesoup_i]
< 'z') {
        stonesoup_global_str[stonesoup_i] -= 'a' - 'A';
    }
    stonesoup_i++;
}
sem_post(&stonesoup_sem);
```

In an exploiting run this thread has the possibility of deadlocking on the call to `sem_wait()` shown above. If the `replaceSpace()` thread had called `sem_wait()` twice, deadlocking on the second call, before this call to `sem_wait()` it will cause this thread to deadlock on that call as well.

replaceSpace()

This thread is also called without any parameters, and once executing makes two calls to `sem_wait()` causing a deadlock. This is the trigger point.

```
/* STONESOUP: TRIGGER-POINT (multiple locks) */
 sem_wait(&stonesoup_sem);                                   /* multiple locks -
deadlock */
 sem_wait(&stonesoup_sem);
```

The rest of this function is irrelevant since the thread will not execute past the second call to `sem_wait()`.

Benign Inputs

Benign inputs do not contain any spaces. For example:

```
aTest
```

```
abigwordwithoutspaces
```

```
anotherTestWithoutSpace
```

Exploiting Inputs

Exploiting inputs all contain spaces. For example:

```
This is a bad test.
```

```
A NEW TEST
```

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Restrictions and Requirements

C - CWE-765A - Multiple Unlocks of a Critical Resource

Summary

C programs have two main options for mutual exclusion of a shared resource in a concurrent setting: the mutex lock and the semaphore.

Mutex locks exist in two states, locked and unlocked. However a semaphore, more specifically a counting semaphore, can be used to keep track of an arbitrary number of resources that are available at any given time, allowing access until all of the resources are being used. Using a counting semaphore, multiple unlocks will cause the semaphore to enter a bad state in which it indicates that there are more resources available than there actually are. This will then allow more threads to access the given resource than are allowed, leading to a bad state.

This weakness variant uses a counting semaphore initialized to one unit of the shared resource. For inputs that contain a capital letter, a function is called that requests a resource, acts upon this resource, and releases the resource twice. This causes the semaphore to erroneously increment its number of units to two. Two threads are then spawned, both of which request the resource. The semaphore erroneously allows both to access the resource, and one thread gets a null pointer dereference. In the benign case, the semaphore allows only one thread to access the resource at a time, and the snippet executes safely.

This weakness is an example of a Multi-Threaded Race Condition.

Weakness Class	Concurrency Handling
CWE	CWE-765
Variant	A

Language	C
Status	WEAK-368 - CWE-765-0-C DONE
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

This snippet contains three main functions, toLower(), reverseStr(), and to1337(). All three functions act upon a shared string (copied from the input) and use a counting semaphore for data synchronization. reverseStr() and to1337() are executed as threads. toLower() is called before the other two functions/threads are spawned, but only in the case of an input string containing a capital letter.

The crossover point resides in toLower() which decrements the semaphore, converts all upper case letters in the string to lower case, and increments the semaphore twice, causing multiple unlocks of a critical resource.

```
sem_post(&stonesoup_sem);
    /* STONESOUP: CROSSOVER-POINT (multipleunlocks) */
    sem_post(&stonesoup_sem);                                  /* oops, extra unlock
*/
```

If toLower() has been called, the semaphore erroneously indicates that two units of the shared resource are available, and both reverseStr() and to1337() are able to execute at the same time, possibly causing a null pointer dereference (based on the input, described below). The trigger point residers in reverseStr(), as it iterates over the characters in shared buffer, creating a new array in reverse order.

```
for (i = 0; i < stonesoupData->data_size; i++) {
        /* STONESOUP: TRIGGER-POINT (multipleunlocks) */
        stonesoup_printf("Dereferencing ptr\n");
        temp_str[stonesoupData->data_size - 1 - i] = stonesoupData->data[i];    /*
null ptr dereference */
    }
```

If toLower() has not been called, the semaphore works as intended and reverseStr() and to1337() are forced to execute with proper mutual exclusion, avoiding the weakness entirely.

This weakness closely follows the general implementation of Multi-Threaded Race Conditions. Weakness Thread W1 uses reverseStr, and Weakness Thread W2 uses to1337. The crossover point occurs in the toLower method, before either thread is spawned. The bad state is created in the to1337 method, when it sets stonesoupData->data = NULL. The trigger point occurs in the reverseStr method, when it accesses stonesoupData->data[i]. If stonesoupData->data is NULL, this access causes a null pointer dereference, which causes a segmentation fault. to1337 returns to a good state when it sets stonesoupData->data = temp_str.

For the sake of clarity, we will assume that any time the the runFifos.py script is required it was executed as follows:

runFifos.py

```
runFifos.py /opt/stonesoup/weakness/testData/fifo1
/opt/stonesoup/weakness/testData/fifo2
```

Benign Inputs

All benign inputs exist in one of the following two forms:

FIFO Files

```
<qsize> <FIFO> <FIFO> <string>
```

If you execute a benign input with FIFO files, you should provide the FIFO files to the snippet in the reverse order from how they were provided to runFifos.py (eg, /opt/.../fifo2 /opt/.../fifo1). The qsize argument is irrelevant to ensuring the successful execution of the benign input, however a large qsize will slow down the overall execution. The string in this case will not affect the overall execution of the snippet, however a string containing a capital letter will cause the compromised code path to be executed, albeit with benign timing due to the external synchronization of the FIFO files.

Examples are as follows:

```
5 /opt/stonesoup/workspace/testData/fifo2 /opt/stonesoup/workspace/testData/fifo1
hephalump
```

This input has no chance of error, because the final input string has no capital letters. This means that the toLower method is never called, and the semaphore is never incorrectly double-decremented. The threads executing to1337 and reverseStr will use correct locking procedures, and execution will always complete successfully.

```
60 /opt/stonesoup/workspace/testData/fifo2 /opt/stonesoup/workspace/testData/fifo1
Sniffles_and_whiffles
```

This input has a theoretical chance of error, because the final input string has capital letters. This means that the toLower method is called, and the semaphore is incorrectly double-decremented. The threads executing to1337 and reverseStr will not use correct locking procedures and will try to access shared data simultaneously. However, the FIFO files are provided in the reverse order from the way they are provided to the runFifos.py script. This external control of timing will almost always allow the execution to take place without error, though there is still the (small) chance of an incorrect schedule.

Regular Files

```
<qsize> <file> <file> <string>
```

When executing the snippet with regular files the order of the files is irrelevant, and the data contained within them is as well, as long as it is trivial in length. The qsize argument is the important argument for ensuring that the weakness executes in a benign manner. When given a string containing a capital letter, the compromised code path is executed, however a large qsize value will almost always delay execution of one of the threads long enough to allow the other thread to access the shared resource while it is in a good state.

Examples are as follows:

```
50 /opt/stonesoup/workspace/testData/file1 /opt/stonesoup/workspace/testData/file2
look_ma,_no_caps
```

This input has no chance of error, because the final input string has no capital letters. This means that the toLower method is never called, and the semaphore is never incorrectly double-decremented. The threads executing to1337 and reverseStr will use correct locking procedures, and execution will always complete successfully.

```
6000000 /opt/stonesoup/workspace/testData/file1
/opt/stonesoup/workspace/testData/file2 sIlly_rabbit,_tricks_are_for_kids.
```

This input has a theoretical chance of error, because the final input string has capital letters. This means that the toLower method is called, and the semaphore is incorrectly double-decremented. The threads executing to1337 and reverseStr will not use correct locking procedures and will try to access shared data simultaneously. The regular files do not provide the external timing control that the runFifos.py script does.

However, the large quicksort size will almost always result in a benign thread schedule.

Exploiting Inputs

Exploiting inputs all use FIFO files to ensure a negative technical impact and are of the form:

```
<qsize> <FIFO1> <FIFO2> <string>
```

Like the FIFO benign inputs, the qsize is irrelevant to the correct behavior of the snippet, however a large qsize value will slow down the overall execution of the snippet. Unlike the FIFO benign inputs, the FIFO files should be specified in the order that they were specified in the runFifos.py script (/opt/.../fifo1 /opt/.../fifo2). The order of these files determines the timing within the file and the string must contain a capital letter in order for the compromised code path to be executed.

Examples are as follows:

```
508 /opt/stonesoup/workspace/testData/fifo1 /opt/stonesoup/workspace/testData/fifo2
May_the_force_be_with_you.
```

```
99 /opt/stonesoup/workspace/testData/fifo1 /opt/stonesoup/workspace/testData/fifo2
What_is_the_answer_to_life,_the_universe,_and_everything?
```

Both these inputs have a capital letter, so toLower is called and the semaphore is incorrectly decremented twice. The threads executing to1337 and reverseStr will not use correct locking procedures and will try to access shared data simultaneously. The FIFO files will influence the schedule such that reverseStr accesses the shared data after to1337 has set it to a bad value (NULL), causing a segmentation fault.

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Concurrency Handling
- Execution Environment
- Multi-Process Race Conditions
- Multi-Threaded Race Conditions
- Restrictions and Requirements

C - CWE-765B - Multiple Unlocks of a Critical Resource

Summary

This snippet uses a counting semaphore initialized to one count of a shared resource to implement multiple unlocks of a critical resource for certain input. The snippet takes a control integer, the names of two control files, and an input string. The control integer and the two control files are used for timing within the snippet to ensure that the snippet follows an exploiting or benign execution path, and the input string is used as shared data for the threads to act upon. When executing, the snippet checks the input string for non-alpha characters, and if it contains non-alpha characters the snippet increments the counting semaphore and spawns two threads that both use the semaphore as an access control mechanism surrounding a shared reference to the input string. Since the semaphore incorrectly indicates that there are two counts of the input string available, both threads now have the ability to access the shared string (actual concurrent access is controlled by the control files and integer) leading to a segfault. If the input string does not contain non-alpha characters, only one thread will spawn and the semaphore will be decremented and incremented appropriately, allowing the snippet to run without error.

This weakness is an example of a Multi-Threaded Race Condition.

Weakness Class	Concurrency Handling
CWE	CWE-765
Variant	B
Language	C
Status	WEAK-407 - CWE-765-1-C DONE
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

This snippet contains two main functions: `toCap()` and `delNonAlpha()`. Both functions are run concurrently by two separate threads or, depending on the input, only `toCap()` is executed by a single thread. The snippet starts execution by saving the input values to a shared struct called 'stonesoupData' that gets passed by reference to both functions and initializing a semaphore to one count of the shared resource. The snippet then iterates through the input string searching for a non-alpha character. If the snippet finds a non-alpha character it increments the semaphore and spawns both threads. The incrementing of the semaphore is the crossover point.

```
sem_init(&stonesoup_sem, 0, 1);
while (stonesoupData->data[stonesoup_i] != '\0') {                    /* parse
input for non-alpha */
    if(stonesoup_isalpha(stonesoupData->data[stonesoup_i]) == 0) {
        hasNonAlpha = 1;
    }
    stonesoup_i++;
}
if (hasNonAlpha != 0) {
    /* STONESOUP: CROSSOVER-POINT (unlockedresourceunlock) */
    sem_post(&stonesoup_sem);
    pthread_create(&stonesoup_t0, NULL, delNonAlpha, stonesoupData);     /* thread
will run concurrently with */
}                                                                /*  next thread due
to the unlock on the semaphore */
pthread_create(&stonesoup_t1, NULL, toCap, stonesoupData);
```

If the snippet did not find a non-alpha character in the input string, the crossover-point will not occur and only the `toCap()` thread will be spawned.

toCap()

The `toCap()` thread is called with a reference to the shared struct containing the input data. Once executing, the thread calls `sem_wait()` and allocates an integer array with size equal to the control integer. The thread then populates the newly created array with integers in descending order (from size to 0) and sorts the array into ascending order. The purpose of the sort is to slow down the execution of the snippet in certain benign executions (explained in more detail in the next section). Once the array sort is finished, the thread reads it's control file (again to control the execution time of the thread, explained below), and upon returning from the call to read the file iterates though the shared input string converting all alpha characters to capitals, calling `sem_post()` when it is finished.

```
sem_wait(&stonesoup_sem);                     /* sem lock fails when extra unlock
occurs */
/* slow things down to make correct thing happen in good cases */
stonesoup_arr = malloc(sizeof(int) * stonesoupData->qsize);
for (stonesoup_i = 0; stonesoup_i < stonesoupData->qsize; stonesoup_i++) {
    stonesoup_arr[stonesoup_i] = stonesoupData->qsize - stonesoup_i;
}
qsort(stonesoup_arr, stonesoupData->qsize, sizeof(int), &stonesoup_comp);
free(stonesoup_arr);
stonesoup_readFile(stonesoupData->file1);
for(i = 0; i < strlen(stonesoupData->data); i++) {
    if (stonesoupData->data[i] >= 'a' && stonesoupData->data[i] <= 'z') {     /*
null pointer dereference when concurrent */
        stonesoupData->data[i] -= 32;                                           /*
with other thread */
    }
}
sem_post(&stonesoup_sem);
```

If the `delNonAlpha()` thread was also spawned, depending on the control files, the `toCap()` thread may segfault while iterating through the shared string due to a possible null pointer dereference.

delNonAlpha()

The `delNonAlpha()` thread is only called if the input string contains a non-alpha character. If this thread is called it is passed a reference to the shared struct that contains the snippets input data. Once executing, the thread calls `sem_wait()` which decrements the erroneously set semaphore (previously incremented in the main thread) from 2 to 1 copies of the shared string, allowing the `toCap()` thread to also call `sem_wait()` and access the shared input string. The thread then iterates through the shared input string, copying each alpha character into a new array. The shared string is then `free()'d` and set to null. The thread then reads its control file, pausing its execution, and if running with exploiting input, allowing the `toCap()` thread enough time to dereference the newly set null pointer, causing a segfault. This is the trigger point.

```
sem_wait(&stonesoup_sem);
temp = malloc(sizeof(char) * (strlen(stonesoupData->data) + 1));
while(stonesoupData->data[i] != '\0') {
    if (stonesoup_isalpha(stonesoupData->data[i])) {
        temp[j++] = stonesoupData->data[i];
    }
    i++;
}
temp[++j] = '\0';
free(stonesoupData->data);
stonesoupData->data = NULL;               /* after this line, other thread runs and
dereferences null pointer */

/* STONESOUP: TRIGGER-POINT (unlockedresourceunlock) */
stonesoup_readFile(stonesoupData->file2);
```

If the control files or control integer pause the execution of the threads in such a way that the null pointer dereference does not occur, then the thread will set the shared string to the temp string and complete execution without error.

Timing Explanation

Due to the random nature of race conditions, we cannot rely on the thread scheduler alone to run our threads and expect that they will

deterministically result in either a good or bad outcome. In order to provide a deterministic execution environment we introduced two control mechanisms into the threads themselves. The first is a set of two control files which, when created as FIFO files and controlled by the runFifos.py script described below, ensure that one file read completes two seconds before the other. The other control method is to pass in two regular files for the control files, essentially negating the readFile() commands, and instead passing a significantly large integer in order to influence the thread timing though an array sort on an array of descending integers with size equal to the input integer.

For more information on either of these approaches, view the documentation for multi-threaded race conditions found here.

runFifos.py

The runFifos.py script indirectly controls the time execution order of the threads in a snippet, the specific configuration described below is set by passing two file names of nonexistent files to the runFifos.py script. The script will create both files and control them to ensure that the first FIFO closes two seconds before the second, allowing them to be used to control the execution of the snippet. This is done by writing 32 bytes of data to both FIFOs and waiting for them both to be read, ensuring that both threads in the snippet are synchronized on the file read. The script then writes a second 32 bytes to the first FIFO file and closes it. It then waits two seconds and writes a second 32 bytes to the second FIFO file and closes it as well. The order is as follows:

```
t1: wrote first 32 bytes to file
t2: wrote first 32 bytes to file
ALL: threads synchronized
t1: wrote second 32 bytes to file
t2: t1 finished
t2: wrote second 32 bytes to file
```

The correct execution of this script for the snippets bad test cases is shown below.

```
runFifos.py /opt/stonesoup/workspace/testData/fifo1
/opt/stonesoup/workspace/testData/fifo2
```

Where `/opt/stonesoup/workspace/testData/fifo1` and `/opt/stonesoup/workspace/testData/fifo2` are the names of the control files that will be passed to the snippet when it is executed.

It is worth noting that the runFifos.py script must be executed as a backgrounded pre-process to the snippets execution and allowed to continue to run during the execution of the snippet.

The source code for this script can be found here.

Benign Inputs

Benign inputs come in two forms. The first is using FIFO files with the order passed to the snippet reversed from the order called in the `runFifos.py` script. When executing benign test cases in this format the value of the input integer does not change the execution of the snippet and only serves to slow the overall execution of the snippet when the value is sufficiently large. The second format is to call the snippet with two regular files as control files and a significantly large control integer.

For the following examples that use FIFO files we will assume that the `runFifos.py` script is executed in the format show above.

FIFO Files

Examples of benign inputs with FIFO control files are as follows:

```
1 /opt/stonesoup/workspace/testData/fifo2 /opt/stonesoup/workspace/testData/fifo1
hephalump!
```

```
60 /opt/stonesoup/workspace/testData/fifo2 /opt/stonesoup/workspace/testData/fifo1
Sniffles_and_whiffles
```

Regular Files

Examples of benign inputs with regular control files are as follows:

```
6000000 /opt/stonesoup/workspace/testData/file1
/opt/stonesoup/workspace/testData/file2 sIllyrabbittricksareforkids
```

Exploiting Inputs

Exploring inputs all use FIFO control files passed in the same order as they are called in the `runFifos.py` script. Like the benign test case using FIFO control files above, the value of the integer input is irrelevant and simply serves to slow the overall execution of the snippet when the value is sufficiently large. For example:

```
508 /opt/stonesoup/workspace/testData/fifo1 /opt/stonesoup/workspace/testData/fifo2
May_the_force_be_with_you.
```

```
99 /opt/stonesoup/workspace/testData/fifo1 /opt/stonesoup/workspace/testData/fifo2
What_is_the_answer_to_life,_the_universe,_and_everything?
```

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Concurrency Handling
- Execution Environment
- Multi-Process Race Conditions
- Multi-Threaded Race Conditions
- Restrictions and Requirements

C - CWE-820A - Missing Synchronization

Summary

This snippet implements two threads that do not use synchronization while accessing a shared resource. The snippet takes a control integer, the names of two control files, and an input string. The control integer and the two control files are used for timing within the snippet to ensure that the snippet follows an exploiting or benign execution path, and the input string is used as shared data for the threads to act upon. When executing, the snippet spawns two threads, one of which calculates an increment amount, and the other uses this increment amount to iterate through the shared input string setting each position to a '#'. If the calculated increment amount is negative, and the thread that uses it to iterate though the string does so before it is sanitized, the snippet will underwrite its buffer, causing a segfault if it underwrites far enough.

This is an example of a Multi-Threaded Race Condition .

Weakness Class	Concurrency Handling

CWE	CWE-820
Variant	A
Language	C
Status	WEAK-371 - CWE-820-0-C DONE
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

This snippet contains two main functions: `calcIncamount()`, and `toPound()`, both of which are run concurrently by two threads. The two functions are described below:

calcIncamount()

The `calcIncamount()` thread is called with a reference to the struct containing the snippet's input data. Once executing, the thread calculates a global increment based on the first character of the input string, and reads the control file (used for thread timing, described in the next section). This is the crossover point.

```
dataStruct->inc_amount = dataStruct->data[0] - 'A';                    /*
oops...um... */
stonesoup_readFile(dataStruct->file2);
```

Once the control file is finished being read, assuming the weakness has not occurred, the thread sanitizes the global increment.

```
if (dataStruct->inc_amount < 0) {                                      /* let's
just clean up and */
    dataStruct->inc_amount *= -1;                                      /*  pretend
that never happened */
}
else if (dataStruct->inc_amount == 0) {                                /*  shhhh */
    dataStruct->inc_amount += 1;
}
```

The thread then completes its execution and joins the main thread without error.

toPound()

The `toPound()` thread is called with a reference to the struct containing the snippet's input data. Once executing, the thread allocates an integer array with size equal to the control integer, and populates the newly created array with integers in descending order (from size to 0). This array is then sorted to ascending order. The purpose of the sort is to slow down the execution of the snippet in certain good test cases (explained in more detail in the next section). Once the array sort is finished, the thread reads it's control file (again to control the execution time of the thread, explained below), and iterates through the shared input string setting each character iterated over to a '#', using the global increment calculated by `calcIncamount()` as the increment for it's loop counter. If the increment is currently set to a negative number this loop will underwrite the buffer, causing a segfault if it underwrites far enough through memory. This is the trigger point.

```
/* slow things down to make correct thing happen in good cases */
stonesoup_arr = malloc(sizeof(int) * dataStruct->qsize);
for (stonesoup_i = 0; stonesoup_i < dataStruct->qsize; stonesoup_i++) {
    stonesoup_arr[stonesoup_i] = dataStruct->qsize - stonesoup_i;
}
qsort(stonesoup_arr, dataStruct->qsize, sizeof(int), &stonesoup_comp);
free(stonesoup_arr);
stonesoup_readFile(dataStruct->file1);

/* STONESOUP: TRIGGER-POINT (missing syncronization) */
for (stonesoup_i = 0; stonesoup_i < (int)strlen(dataStruct->data) - 1;
     stonesoup_i += dataStruct->inc_amount)                            /* can
cause underread/write if */
{
    dataStruct->data[stonesoup_i] = '#';                               /*
stonesoup_increment_amount is neg */
}
```

If the increment is positive, the loop will overwrite the string properly and the thread will complete execution, joining the main thread without error.

Timing Explanation

Due to the random nature of race conditions, we cannot rely on the thread scheduler alone to run our threads and expect that they will deterministically result in either a good or bad outcome. In order to provide a deterministic execution environment we introduced two control mechanisms into the threads themselves. The first is a set of two control files which, when created as FIFO files and controlled by the runFifos.py script described below, ensure that one file read completes two seconds before the other. The other control method is to pass in two regular files for the control files, essentially negating the readFile() commands, and instead passing a significantly large integer in order to influence the thread timing though an array sort on an array of descending integers with size equal to the input integer.

For more information on either of these approaches, view the documentation for multi-threaded race conditions found here.

runFifos.py

The runFifos.py script indirectly controls the time execution order of the threads in a snippet, the specific configuration described below is set by passing two file names of nonexistent files to the runFifos.py script. The script will create both files and control them to ensure that the first FIFO closes two seconds before the second, allowing them to be used to control the execution of the snippet. This is done by writing 32 bytes of data to both FIFOs and waiting for them both to be read, ensuring that both threads in the snippet are synchronized on the file read. The script then writes a second 32 bytes to the first FIFO file and closes it. It then waits two seconds and writes a second 32 bytes to the second FIFO file and closes it as well. The order is as follows:

```
t1: wrote first 32 bytes to file
t2: wrote first 32 bytes to file
ALL: threads synchronized
t1: wrote second 32 bytes to file
t2: t1 finished
t2: wrote second 32 bytes to file
```

The correct execution of this script for the snippets bad test cases is shown below.

```
runFifos.py /opt/stonesoup/workspace/testData/fifo1
/opt/stonesoup/workspace/testData/fifo2
```

Where `/opt/stonesoup/workspace/testData/fifo1` and `/opt/stonesoup/workspace/testData/fifo2` are the names of the control files that will be passed to the snippet when it is executed.

The runFifos.py script must be executed as a backgrounded pre-process to the snippet's execution and allowed to continue to run during the execution of the snippet.

The source code for this script can be found here.

Benign Inputs

Benign inputs come in two forms. The first is using FIFO files with the order passed to the snippet reversed from the order called in the `runFifos.py` script. When executing benign test cases in this format the value of the input integer does not change the execution of the snippet and only serves to slow the overall execution of the snippet when the value is sufficiently large. The second format is to call the snippet with two regular files as control files and a significantly large control integer.

For the following examples that use FIFO files we will assume that the `runFifos.py` script is executed in the format show above.

FIFO Files

Examples of benign inputs with FIFO control files are as follows:

```
5 /opt/stonesoup/workspace/testData/fifo2 /opt/stonesoup/workspace/testData/fifo1
Dirt_McGert
```

```
1 /opt/stonesoup/workspace/testData/fifo2 /opt/stonesoup/workspace/testData/fifo1
@the_longest_hashtag_that_ever_existed_on_the_face_of_the_earth
```

Regular Files

Examples of benign inputs with regular control files are as follows:

```
6000000 /opt/stonesoup/workspace/testData/file1
/opt/stonesoup/workspace/testData/file2 Silly_rabbit,_tricks_are_for_kids
```

Exploiting Inputs

Exploring inputs all use FIFO control files passed in the same order as they are called in the `runFifos.py` script. Like the benign test case using FIFO control files above, the value of the integer input is irrelevant and simply serves to slow the overall execution of the snippet when the value is sufficiently large. For example:

```
508 /opt/stonesoup/workspace/testData/fifo1 /opt/stonesoup/workspace/testData/fifo2
@the_longest_hashtag_that_ever_existed_on_the_face_of_the_earth
```

```
99 /opt/stonesoup/workspace/testData/fifo1 /opt/stonesoup/workspace/testData/fifo2
1234567890_0987654321_1029384756_5647382910_0192837465_6574839201
```

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Restrictions and Requirements

C - CWE-821A - Incorrect Synchronization

Summary

This snippet implements two threads that each use a separate mutex lock object while accessing a shared resource. The snippet takes a control integer, the names of two control files, and an input string. The control integer and the two control files are used for timing within the snippet to ensure that the snippet follows an exploiting or benign execution path, and the input string is used as shared data for the threads to act upon. When executing, the snippet spawns two threads, one of which calculates a divisor based on the input string, and the other of which uses that divisor to divide each character in the string. If the divisor is calculated to be 0 and the second thread uses this value before it is san itized the snippet segfault with a divide by zero error.

This is an example of a Multi-Threaded Race Condition .

Weakness Class	Concurrency Handling
CWE	CWE-821
Variant	A
Language	C
Status	WEAK-375 - CWE-821-0-C DONE
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

This snippet contains two main functions: `calcDevamount()` and `devChar()`, which are both run concurrently by two separate threads. The two functions are described below:

calcDevAmount()

The `calcDevAmount()` thread is called with a reference to the shared struct containing the input data. Once executing, the thread locks a mutex lock called 'stonesoup_mutex_0' and calculates a divisor from the first character of the input string. This is the crossover point.

```
pthread_mutex_lock(&stonesoup_mutex_0);

/* STONESOUP: CROSSOVER-POINT (incorrect syncronization) */
stonesoup_dev_amount = stonesoupData->data[0] - 'A';                    /*
oops...um... */
```

The thread then partially sanitizes the data and reads the control file (used to control the thread execution time, explained in the next section), and once the control file read is finished, assuming the weakness has not been executed, the thread finishes sanitizing the divisor.

```
qsize = stonesoupData->qsize;
if (stonesoup_dev_amount < 0) {                                          /* let's
just clean up and */
    stonesoup_dev_amount *= -1;                                          /*
pretend that never happened */
}

stonesoup_readFile(stonesoupData->file2);

if (stonesoup_dev_amount == 0) {                                         /* shhhh,
just some more cleanup */
    stonesoup_dev_amount += 1;                                           /*
nothing to see here */
}
```

Assuming the weakness did not occur, the thread will then complete its execution, joining the main thread without error.

devChar()

The `devChar()` thread is called with a reference to the shared struct containing the input data. Once executing, the thread locks a second mutex lock called 'stonesoup_mutex_1' and allocates an integer array with size equal to the control integer, populating the newly created array with integers in descending order (from size to 0). This array is then sorted to ascending order. The purpose of the sort is to slow down the execution of the snippet in certain benign executions (explained in more detail in the next section). Once the array sort is finished, the thread reads it's control file (again to control the execution time of the thread, explained below), and iterates through the shared string dividing each character with the divisor calculated by the other thread. If the divisor was calculated as zero and not yet sanitized by the other thread this will result in a divide by zero error. This is the trigger point.

```
pthread_mutex_lock(&stonesoup_mutex_1);
for (stonesoup_i = 0; stonesoup_i < stonesoupData->qsize; stonesoup_i++) {
    stonesoup_arr[stonesoup_i] = stonesoupData->qsize - stonesoup_i;
}
qsort(stonesoup_arr, stonesoupData->qsize, sizeof(int), &stonesoup_comp);
free(stonesoup_arr);
stonesoup_readFile(stonesoupData->file1);

/* STONESOUP: TRIGGER-POINT (incorrect syncronization) */
for (i = 0; i < strlen(stonesoupData->data); i++) {                      /* can
cause underread/write if */
    stonesoupData->data[i] /= stonesoup_dev_amount;                      /*
stonesoup_dev_amount is neg */
}
```

If the divisor is a positive non-zero number, the thread will finish executing and join the main thread without error.

Timing Explanation

Due to the random nature of race conditions, we cannot rely on the thread scheduler alone to run our threads and expect that they will deterministically result in either a good or bad outcome. In order to provide a deterministic execution environment we introduced two control mechanisms into the threads themselves. The first is a set of two control files which, when created as FIFO files and controlled by the runFifos.py script described below, ensure that one file read completes two seconds before the other. The other control method is to pass in two regular files for the control files, essentially negating the readFile() commands, and instead passing a significantly large integer in order to influence the thread timing though an array sort on an array of descending integers with size equal to the input integer.

For more information on either of these approaches, view the documentation for multi-threaded race conditions found here.

runFifos.py

The runFifos.py script indirectly controls the time execution order of the threads in a snippet, the specific configuration described below is set by passing two file names of nonexistent files to the runFifos.py script. The script will create both files and control them to ensure that the first FIFO closes two seconds before the second, allowing them to be used to control the execution of the snippet. This is done by writing 32 bytes of data to both FIFOs and waiting for them both to be read, ensuring that both threads in the snippet are synchronized on the file read. The script then writes a second 32 bytes to the first FIFO file and closes it. It then waits two seconds and writes a second 32 bytes to the second FIFO file and closes it as well. The order is as follows:

```
t1: wrote first 32 bytes to file
t2: wrote first 32 bytes to file
ALL: threads synchronized
t1: wrote second 32 bytes to file
t2: t1 finished
t2: wrote second 32 bytes to file
```

The correct execution of this script for the snippets bad test cases is shown below.

```
runFifos.py /opt/stonesoup/workspace/testData/fifo1
/opt/stonesoup/workspace/testData/fifo2
```

Where `/opt/stonesoup/workspace/testData/fifo1` and `/opt/stonesoup/workspace/testData/fifo2` are the names of the control files that will be passed to the snippet when it is executed.

The runFifos.py script must be executed as a backgrounded pre-process to the snippet's execution and allowed to continue to run during the execution of the snippet.

The source code for this script can be found here.

Benign Inputs

Benign inputs come in two forms. The first is using FIFO files with the order passed to the snippet reversed from the order called in the `runFifos.py` script. When executing benign test cases in this format the value of the input integer does not change the execution of the snippet and only serves to slow the overall execution of the snippet when the value is sufficiently large. The second format is to call the snippet with two regular files as control files and a significantly large control integer.

For the following examples that use FIFO files we will assume that the `runFifos.py` script is executed in the format show above.

FIFO Files

Examples of benign inputs with FIFO control files are as follows:

```
5 /opt/stonesoup/workspace/testData/fifo2 /opt/stonesoup/workspace/testData/fifo1
Another_test_case
```

```
1 /opt/stonesoup/workspace/testData/fifo2 /opt/stonesoup/workspace/testData/fifo1
hello_world
```

Regular Files

Examples of benign inputs with regular control files are as follows:

```
6000000 /opt/stonesoup/workspace/testData/file1
/opt/stonesoup/workspace/testData/file2 1,_2,_skip_to_my_lou

Where /opt/stonesoup/workspace/testData/file1 and
/opt/stonesoup/workspace/testData/file2 are pre-existing empty files.
```

Exploiting Inputs

Exploring inputs all use FIFO control files passed in the same order as they are called in the `runFifos.py` script, and must have the character of the input string as an 'A'. Like the benign test case using FIFO control files above, the value of the integer input is irrelevant and simply serves to slow the overall execution of the snippet when the value is sufficiently large. For example:

```
508 /opt/stonesoup/workspace/testData/fifo1 /opt/stonesoup/workspace/testData/fifo2
As_he_lay_watching_the_fireflies_over_head,_he_couldn't_shake_the_feeling_that_maybe
_it_was_they_who_were_watching_him.
```

```
99 /opt/stonesoup/workspace/testData/fifo1 /opt/stonesoup/workspace/testData/fifo2
Answering_question_after_question,_our_unlikely_hero_was_finally_winning_the_scholas
tic_challenge_for_us.
```

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Restrictions and Requirements

C - CWE-828A - Signal Handler with Functionality that is not Asynchronous-Safe

Summary

This snippet implements an asynchronous unsafe signal handler that access a string without properly null checking the pointer. The snippet takes the name of a control file and an input string. The control file is used for timing within the snippet to ensure that the snippet follows an exploiting or benign execution path, and the input string is used as shared data for the threads to act upon. When executing, the snippet assigns a signal hander that access an internal array, printing it's data to the output string. The snippet then iterates through the input string, copying each character into the internal array, initialized to a size of 51. If the input string is greater than 50 characters in length, the snippet resizes the array, temporarily setting it to null. If the signal handler is invoked while the new array is set to null the signal handler will dereference the null pointer resulting in a segfault.

This is an example of a Multi-Process Race Condition.

Weakness Class	Concurrency Handling
CWE	CWE-828

Variant	A
Language	C
Status	WEAK-377 - CWE-828-0-C DONE
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

This snippet implements a signal handler that accesses an internal array without proper null checking. The snippet starts execution by assigning a signal handler for the 'SIGUSR1' signal and allocating an internal character array of size 51.

```
if (signal(SIGUSR1, sig_handler) == SIG_ERR) {
    stonesoup_printf ("Error catching SIGNUSR1!\n");
}
playful_platypus = malloc(sizeof(char) * (SIZE + 1));
stonesoup_i = 0;
```

Once the character array is allocated, the snippet iterates though the input string copying each character into the internal array. If there internal array fills up and there are more characters to read the snippet allocates a new array with double the capacity and copies the internal array into it. It then free()s the internal array and sets its pointer to null, reading the control file after. This call to read the control file allows the execution to slow down enough in an exploiting run that an external script (described below) can signal the snippet and cause the signal handler to dereference the null pointer causing a segfault.

```
stonesoup_i = 0;
while (stonesoupData->data[stonesoup_i] != '\0') {                          /* copy
input to global char* */
    if (stonesoup_i < SIZE) {
        playful_platypus[stonesoup_i] = stonesoupData->data[stonesoup_i];
        stonesoup_i++;
    } else {                                                   /* if input size > 50
char, realloc size by hand */
        playful_platypus[SIZE] = '\0';

        /* STONESOUP: CROSSOVER-POINT (asyncunsafesighandler) */
        SIZE *= 2;
        temp = malloc(sizeof(char) * SIZE);
        strcpy(temp, playful_platypus);
        free(playful_platypus);
        playful_platypus = NULL;                            /* calling sig handler after
this instruction to break */
        waitForSig(sleepFile);
  playful_platypus = temp;
 }
}
```

The trigger point itself lies in the signal hander which is shown below.

```
void sig_handler (int sig) {
    stonesoup_printf("In sig_handler\n");

    /* STONESOUP: TRIGGER-POINT (asyncunsafesighandler) */
    /* iterate through array and do something */
    if (playful_platypus[0] != '\0') {                      /* bad error checking -
can cause null ptr deref */
        stonesoup_printf(playful_platypus);
    }
}
```

If the control file was a regular file, and therefore didn't slow the execution enough for the external script to signal the snippet, then the snippet will finish copying the input string to the internal buffer and exit without error.

External Scripts

The execution of this snippet in the bad cases is reliant on two external scripts which will be described in detail below.

runFifos.py

The runFifos.py script indirectly controls the snippets execution time. When run in the configuration required by this snippet, set by passing the script the --nosync flag, the script creates a FIFO file, writes 32 bytes of data to the file, and waits for it to be read. Once the first 32 bytes have been read, the script sleeps for 2 seconds and writes another 32 bytes to the file. After the second 32 bytes have been read the snippet closes the file and deletes it.

This script is able to control the amount of time between setting the bad state in the snippet due to the fact that FIFO files are blocking by default. The readFile call shown above will not return until the FIFO file is closed by the script, which has the file open for writing. This creates a robust way for influencing the execution time for the snippet.

The correct execution of this script for the snippets bad test cases is shown below.

```
runFifos.py /opt/stonesoup/workspace/testData/fifo1 --nosync
```

Where /opt/stonesoup/workspace/testData/fifo1 is the name of the control file that will be passed to the snippet when it is executed.

The runFifos.py script must be executed as a backgrounded pre-process to the snippets execution, and allowed to continue during the execution of the snippet.

The source code for this script can be found here.

inotifywait

The inotifywait script is a one line bash script using the inotify-tools inotifywait command to check for the presence of a specific file created by the snippet, and upon finding that file, use it to identify the PID to signal. This script is essential to the proper functioning of the bad test cases.

The inotifywait script exists in the following form (expanded to multiple lines for clarity):

```
inotifywait -m -e create --format \"%f\" /opt/stonesoup/workspace/testData/ | while
read NEW_FILE; do
 if [[ \"${NEW_FILE}\" == *.pid ]]; then
  PROC_ID=\"${NEW_FILE%.pid}\";
  echo \"Signaling: \" \"${PROC_ID}\" && echo \"Waiting\" && sleep 0.5 && kill -s 10
\"${PROC_ID}\" && echo \"Signaled.\";
 else
  echo \"Ignoring: \" \"/opt/stonesoup/workspace/testData/${NEW_FILE}\";
 fi;
done;
```

The script waits for the snippet to create a file in /opt/stonesoup/workspace/testData/ following the pattern <PID>.pid where <PID> is the PID of the snippet. Once the file has been detected the script sleeps for 0.5 seconds and sends a 'SIGUSR1' signal to the process.

The inotifywait script is run for each test case, good and bad, in order to more properly simulate the race condition present in the snippet. Because of this, the inotifywait utilizes a 0.5 second sleep before signaling the snippet in order to provide the snippet enough time to complete execution in a benign run, but still allow it to signal the snippet in the 2 second window allowed by the runFifos.py script in an exploiting run.

The inotifywait script must be executed as a backgrounded pre-process to the snippets execution and allowed to continue during the execution of the snippet.

Benign Inputs

Benign inputs all use an empty regular file for a control file in order to ensure that the snippet completes execution before the inotifywait script signals it. For example:

```
/opt/stonesoup/workspace/testData/file1
What_is_the_answer_to_life,_the_universe,_and_everything?

Where /opt/stonesoup/workspace/testData/file1 is a pre-existing empty file.
```

```
/opt/stonesoup/workspace/testData/file1
This_string_is_most_definitely_longer_than_50_char,_I_wonder_what_will_happen?

Where /opt/stonesoup/workspace/testData/file1 is a pre-existing empty file.
```

```
/opt/stonesoup/workspace/testData/file1 Silly_rabbit,_tricks_are_for_kids!

Where /opt/stonesoup/workspace/testData/file1 is a pre-existing empty file.
```

Exploiting Inputs

Exploring inputs all use FIFO files controlled by runFifos.py as described above and use an input string longer than 50 characters to ensure that the snippet reallocates the internal array, priming it for exploitation. For example:

```
/opt/stonesoup/workspace/testData/fifo1
This_string_is_most_definitely_longer_than_50_char,_I_wonder_what_will_happen?
```

```
/opt/stonesoup/workspace/testData/fifo1
What_is_the_answer_to_life,_the_universe,_and_everything?
```

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Restrictions and Requirements

C - CWE-831A - Signal Handler Function Associated with Multiple Signals

Summary

This snippet implements a single signal handler that is associated with two signals. The snippet takes the names of two control files and an input string. The control files are used for timing within the snippet to ensure that the snippet follows an exploiting or benign execution path, and the input string is used as shared data for the threads to act upon. When executing, the snippet assigns a signal hander to catch 'SIGURS1' that free()s an internal array, and if the input string starts with an 'A' assigns the same signal handler to catch 'SIGURS2'. If the snippet receives both signals after registering the signal handler to both it will double free() the internal array causing a segfault.

This is an example of a Multi-Process Race Condition using two control files, both of which operate the same way as the one illustrated in the documentation.

Weakness Class	Concurrency Handling
CWE	CWE-831
Variant	A
Language	C
Status	WEAK-381 - CWE-831-0-C DONE
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

This snippet implements a signal handler, registered to two separate signals, that free()s an internal array. The snippet starts execution by assigning a signal handler to the 'SIGUSR1' signal and if the input string starts with an 'A', assigning the same signal handler to the 'SIGUSR2' signal. The snippet then reads each control file in succession.

Approved for public release; distribution unlimited.

```
/* optionally set up sig handler bassed on input */
if (signal(SIGUSR1, stonesoup_sig_handler) == SIG_ERR) {
    stonesoup_printf ("Error catching SIGUSR1!\n");
}
stonesoup_printf("Set up SIGUSR1 handler\n");
if (stonesoupData->data[0] >= 'A' && stonesoupData->data[0] <= 'Z') {
    if (signal(SIGUSR2, stonesoup_sig_handler) == SIG_ERR) {
        stonesoup_printf ("Error catching SIGUSR2!\n");
    }
    stonesoup_printf("Set up SIGUSR2 handler\n");
}
waitForSig();
```

Once the snippet is in `waitForSig()` if the snippet was called with FIFO files as input files, it will pause for 2 seconds per file, allowing time for an external script to send both signals in succession, causing the snippet to call the signal handler, `free()` the internal array, and re-call the signal handler, attempt to `free()` the already `free()`'d internal array and segfault. This free() is both the crossover and the trigger point.

```
void stonesoup_sig_handler (int sig) {
    stonesoup_printf("In stonesoup_sig_handler\n");

    /* STONESOUP: CROSSOVER-POINT (signal handler for multiple signals) */
    /* STONESOUP: TRIGGER-POINT (signal handler for multiple signals) */
    stonesoup_global1[0] = -1;
    free(stonesoup_global1);
    stonesoup_global1 = NULL;

    stonesoup_printf("In sig handler");
}
```

If the snippet was called with regular files for control files it will return from `waitForSig()` before the external script has had time to signal the snippet, finishing execution without error.

External Scripts

The execution of this snippet in an exploiting run is reliant on two external scripts which will be described in detail below.

runFifos.py

The runFifos.py script indirectly controls the execution time for the snippet. When run in the configuration required by this snippet, set by passing the script the --nosync flag, the script creates two FIFO files and for each file writes 32 bytes of data to the file, and waits for it to be read. Once the first 32 bytes have been read, the script sleeps for 2 seconds and writes another 32 bytes to the file. After the second 32 bytes have been read the snippet closes the file and deletes it.

This script is able to control the amount of time the snippet takes to read the file due to the fact that FIFO files are blocking by default. The readFile call shown above will not return until the FIFO file is closed by the script, which has the file open for writing. This creates a robust way for influencing the execution time for the snippet.

The correct execution of this script for the snippets bad test cases is shown below.

```
runFifos.py /opt/stonesoup/workspace/testData/fifo1
/opt/stonesoup/workspace/testData/fifo2 --nosync
```

Where /opt/stonesoup/workspace/testData/fifo1 and /opt/stonesoup/workspace/testData/fifo2 are the names of the control files that will be passed to the snippet when it is executed.

The runFifos.py script must be executed as a backgrounded pre-process to the snippets execution, and allowed to continue during the execution of the snippet.

The source code for this script can be found here.

inotifywait

The inotifywait script is a one line bash script using the inotify-tools inotifywait command to check for the presence of a specific file created by the snippet, and upon finding that file, use it to identify the PID to signal. This script is essential to the proper functioning of the bad test cases.

The inotifywait script exists in the following form (expanded to multiple lines for clarity):

```
inotifywait -m -e create --format \"%f\" /opt/stonesoup/workspace/testData/ | while
read NEW_FILE; do
 if [[ \"${NEW_FILE}\" == *.pid ]]; then
  PROC_ID=\"${NEW_FILE%.pid}\";
  echo \"Signaling: \" \"${PROC_ID}\" && \
  sleep 1 && kill -s 10 \"${PROC_ID}\" && echo \"Signaled (10).  Sleeping...\" && \
  sleep 1 && kill -s 12 \"${PROC_ID}\" && echo \"Signaled (12).\";
 else
  echo \"Ignoring: \" \"/opt/stonesoup/workspace/testData/${NEW_FILE}\";
 fi;
done;
```

The script waits for the snippet to create a file in /opt/stonesoup/workspace/testData/ following the pattern <PID>.pid where <PID> is the PID of the snippet. Once the file has been detected the script sleeps for 1 second and sends a 'SIGUSR1' signal to the process. The script then sleeps for another second and sends the snippet a 'SIGUSR2'.

The inotifywait script is run for each test case, good and bad, in order to more properly simulate the race condition present in the snippet. Because of this, the inotifywait utilizes a 0.5 second sleep before signaling the snippet in order to provide the snippet enough time to complete execution in a benign run, but still allow it to signal the snippet in the 4 second window allowed by the runFifos.py script in an exploiting run.

The inotifywait script must be executed as a backgrounded pre-process to the snippets execution and allowed to continue during the execution of the snippet.

Benign Inputs

Benign inputs all use an empty regular file for a control file in order to ensure that the snippet completes execution before the inotifywait script signals it. For example:

```
/opt/stonesoup/workspace/testData/file1 /opt/stonesoup/workspace/testData/file2
ffddssaa448

Where /opt/stonesoup/workspace/testData/file1 and
/opt/stonesoup/workspace/testData/file2 are pre-existing empty files.
```

```
/opt/stonesoup/workspace/testData/file1 /opt/stonesoup/workspace/testData/file2
77dasd

Where /opt/stonesoup/workspace/testData/file1 and
/opt/stonesoup/workspace/testData/file2 are pre-existing empty files.
```

```
/opt/stonesoup/workspace/testData/file1 /opt/stonesoup/workspace/testData/file2
What_do_we_have_here?

Where /opt/stonesoup/workspace/testData/file1 and
/opt/stonesoup/workspace/testData/file2 are pre-existing empty files.
```

Exploiting Inputs

Exploring inputs all use FIFO files controlled by runFifos.py as described above, allowing the inotifyscript enough time to signal the snippet. For example:

```
/opt/stonesoup/workspace/testData/fifo1 /opt/stonesoup/workspace/testData/fifo2
I_do_not_even_what?
```

```
/opt/stonesoup/workspace/testData/fifo1 /opt/stonesoup/workspace/testData/fifo2
Yes,_ma'am!
```

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Restrictions and Requirements

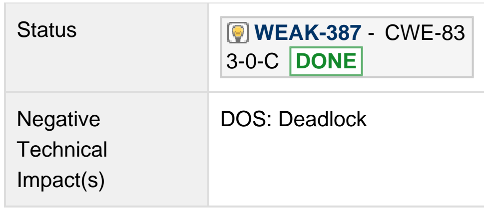
C - CWE-833A - Deadlock

Summary

This snippet implements two thread that both lock two shared mutex locks such that if the timing works out, they will cause each other to deadlock.. The snippet takes a control integer, the names of two control files, and an input string. The control integer and the two control files are used for timing within the snippet to ensure that the snippet follows an exploiting or benign execution path, and the input string is used as shared data for the threads to act upon. When executing with exploiting input the snippet spawns two threads, the first thread (thread A) starts by pausing its execution while the second thread (thread B) locks the second mutex lock. Thread B then pauses its execution and allows thread A to lock the first mutex lock and try to lock the second mutex lock. Thread A will now hang on the call to lock the second mutex lock since thread B holds it, thread B will then try to grab the first mutex lock, but since thread A holds it the system will enter deadlock.

This is an example of a Multi-Threaded Race Condition.

Weakness Class	Concurrency Handling
CWE	CWE-833
Variant	A
Language	C



Status	WEAK-387 - CWE-833-0-C DONE
Negative Technical Impact(s)	DOS: Deadlock

Implementation

This snippet contains two main functions: `stonesoup_replace()` and `stonesoup_toCap()`, which are both run concurrently by two separate threads. The snippet starts by checking the input string for underscores, and the input string contains underscores both threads are spawned, otherwise only `stonesoup_toCap()` is spawned, removing the possibility of deadlock entirely.

```
while(stonesoupData->data[stonesoup_i] != '\0') {                   /* if the input
contains underscores */
    if (stonesoupData->data[stonesoup_i++] == '_') {                /*   we call the
deadlocking function */
        stonesoup_hasUnderscores = 1;
    }
}
if (pthread_create(&stonesoup_t0, NULL, stonesoup_toCap, stonesoupData) != 0) {
    stonesoup_printf("Thread 0 failed to spawn.");
}
if (stonesoup_hasUnderscores == 1) {
    /* STONESOUP: CROSSOVER-POINT (deadlock) */
    if (pthread_create(&stonesoup_t1, NULL, stonesoup_replace, stonesoupData) != 0)
{
        stonesoup_printf("Thread 1 failed to spawn.");
    }
}
```

The two main functions are described below:

replace: SLEEP A B

toCap: B SLEEP A

stonesoup_replace()

The `stonesoup_replace()` thread is called with a reference to the shared struct containing the input data. Once executing, allocates an integer array with size equal to the control integer, populating the newly created array with integers in descending order (from size to 0). This array is then sorted to ascending order. The purpose of the sort is to slow down the execution of the snippet in certain benign executions (explained in more detail in the next section). Once the array sort is finished, the thread reads it's control file (again to control the execution time of the thread, explained below), and once the file read is finished grabs the 'stonesoup_mutex_0' mutex lock. The thread then attempts to grab the 'stonesoup_mutex_1' mutex lock. If the snippet was called with exploiting input this call to lock the mutex will not return since the `stonesoup_toCap()` thread will have already locked 'stonesoup_mutex_1'.

```
/* slow things down to make correct thing happen in good cases */
qsort_arr = malloc(sizeof(int)*stonesoupData->qsize);
if (qsort_arr != NULL) {
    for (i = 0; i < stonesoupData->qsize; i++) {
        qsort_arr[i] = stonesoupData->qsize - i;
    }
    qsort(qsort_arr, stonesoupData->qsize, sizeof(int), &stonesoup_comp);
    free (qsort_arr);
    qsort_arr = NULL;
}
stonesoup_readFile(stonesoupData->file1);
stonesoup_printf("replace: Attempting to grab lock 0\n");
pthread_mutex_lock(&stonesoup_mutex_0);
stonesoup_printf("replace: Grabbed lock 0\n");

stonesoup_printf("replace: Attempting to grab lock 1\n");
pthread_mutex_lock(&stonesoup_mutex_1);                             /* DEADLOCK */
stonesoup_printf("replace: Grabbed lock 1\n");
```

If the snippet was called with benign inputs, the snippet would properly grab both mutex locks, replace all '_' with '-' in the input string, and release both mutex locks, exiting without error.

```
i = 0;
while(stonesoupData->data[i] != '\0') {
    if (stonesoupData->data[i] == '_') {
        stonesoupData->data[i] = '-';
    }
    i++;
}
stonesoup_printf("replace: Releasing lock 1\n");
pthread_mutex_unlock(&stonesoup_mutex_1);
stonesoup_printf("replace: Releasing lock 0\n");
pthread_mutex_unlock(&stonesoup_mutex_0);
```

stonesoup_toCap()

The `stonesoup_toCap()` thread is called with a reference to the shared struct containing the input data. Once executing, the thread grabs the 'stonesoup_mutex_1' mutex lock and reads it's control file. If the snippet was called with exploiting input, the other thread will grab the 'stonesoup_mutex_0' lock while this thread reads it's control file. This snippet then attempts to grab the 'stonesoup_mutex_0' lock, and in the case of exploiting input, will never return from the call to `mutex_lock()` since both threads are waiting on a mutex lock held by the other.

```
stonesoup_printf("toCap:   Entering function\n");
stonesoup_printf("toCap:   Attempting to grab lock 1\n");
pthread_mutex_lock(&stonesoup_mutex_1);
stonesoup_printf("toCap:   Grabbed lock 1\n");
stonesoup_readFile(stonesoupData->file2);
/* STONESOUP: TRIGGER-POINT (deadlock) */
stonesoup_printf("toCap:   Attempting to grab lock 0\n");
pthread_mutex_lock(&stonesoup_mutex_0);                             /* DEADLOCK */
stonesoup_printf("toCap:   Grabbed lock 0\n");
```

If the snippet was called with benign inputs, the snippet will properly grab both mutex locks, convert the input string to capitals, and release

the locks, exiting without error.

```
i = 0;
while(stonesoupData->data[i] != '\0') {
    if (stonesoupData->data[i] > 'a' && stonesoupData->data[i] < 'z') {
        stonesoupData->data[i] -= 'a' - 'A';
    }
    i++;
}
stonesoup_printf("toCap:   Releasing lock 0\n");
pthread_mutex_unlock(&stonesoup_mutex_0);
stonesoup_printf("toCap:   Releasing lock 1\n");
pthread_mutex_unlock(&stonesoup_mutex_1);
```

Timing Explanation

Due to the random nature of race conditions, we cannot rely on the thread scheduler alone to run our threads and expect that they will deterministically result in either a good or bad outcome. In order to provide a deterministic execution environment we introduced two control mechanisms into the threads themselves. The first is a set of two control files which, when created as FIFO files and controlled by the runFifos.py script described below, ensure that one file read completes two seconds before the other. The other control method is to pass in two regular files for the control files, essentially negating the readFile() commands, and instead passing a significantly large integer in order to influence the thread timing though an array sort on an array of descending integers with size equal to the input integer.

For more information on either of these approaches, view the documentation for multi-threaded race conditions found here.

runFifos.py

The runFifos.py script indirectly controls the time execution order of the threads in a snippet, the specific configuration described below is set by passing two file names of nonexistent files to the runFifos.py script. The script will create both files and control them to ensure that the first FIFO closes two seconds before the second, allowing them to be used to control the execution of the snippet. This is done by writing 32 bytes of data to both FIFOs and waiting for them both to be read, ensuring that both threads in the snippet are synchronized on the file read. The script then writes a second 32 bytes to the first FIFO file and closes it. It then waits two seconds and writes a second 32 bytes to the second FIFO file and closes it as well. The order is as follows:

```
t1: wrote first 32 bytes to file
t2: wrote first 32 bytes to file
ALL: threads synchronized
t1: wrote second 32 bytes to file
t2: t1 finished
t2: wrote second 32 bytes to file
```

The correct execution of this script for the snippets bad test cases is shown below.

```
runFifos.py /opt/stonesoup/workspace/testData/fifo1
/opt/stonesoup/workspace/testData/fifo2
```

Where `/opt/stonesoup/workspace/testData/fifo1` and `/opt/stonesoup/workspace/testData/fifo2` are the names of the control files that will be passed to the snippet when it is executed.

The runFifos.py script must be executed as a backgrounded pre-process to the snippet's execution and allowed to continue to run during the execution of the snippet.

The source code for this script can be found here.

Benign Inputs

Benign inputs come in two forms. The first is using FIFO files with the order passed to the snippet reversed from the order called in the `runFifos.py` script. When executing benign test cases in this format the value of the input integer does not change the execution of the snippet and only serves to slow the overall execution of the snippet when the value is sufficiently large. The second format is to call the snippet with two regular files as control files and a significantly large control integer.

For the following examples that use FIFO files we will assume that the `runFifos.py` script is executed in the format show above.

FIFO Files

Examples of benign inputs with FIFO control files are as follows:

```
1 /opt/stonesoup/workspace/testData/fifo2 /opt/stonesoup/workspace/testData/fifo1
a_test
```

```
60 /opt/stonesoup/workspace/testData/fifo2 /opt/stonesoup/workspace/testData/fifo1
abigwordwithoutunderscores
```

Regular Files

Examples of benign inputs with regular control files are as follows:

```
6000000 /opt/stonesoup/workspace/testData/file1
/opt/stonesoup/workspace/testData/file2 another_test

Where /opt/stonesoup/workspace/testData/file1 and
/opt/stonesoup/workspace/testData/file2 are pre-existing empty files.
```

Exploiting Inputs

Exploring inputs all use FIFO control files passed in the same order as they are called in the `runFifos.py` script, and must have the character of the input string as an 'A'. Like the benign test case using FIFO control files above, the value of the integer input is irrelevant and simply serves to slow the overall execution of the snippet when the value is sufficiently large. For example:

```
508 /opt/stonesoup/workspace/testData/fifo1 /opt/stonesoup/workspace/testData/fifo2
This_is_a_bad_test.
```

```
99 /opt/stonesoup/workspace/testData/fifo1 /opt/stonesoup/workspace/testData/fifo2
A_NEW_TEST
```

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Restrictions and Requirements

J - CWE-363A - Race Condition Enabling Link Following

Summary

This snippet implements a time of check time of use vulnerability that allows arbitrary link following. This snippet uses the approach outlined in the Multi-Process Race Conditions page.

The snippet takes in a control file and an input file. The input file is checked to see if it is in the current working directory and not a symbolic link. If both of these conditions are true, the snippet opens and reads the FIFO file and then opens, reads, and prints the data in the input file. Since there is a delay between checking the validity of the file and opening it, the file can be replaced with a symbolic link, and the snippet can be tricked into following it. The control file is used as a method to delay execution long enough for this vulnerability to occur.

This is an example of a Multi-Process Race Condition.

This weakness differs from CWE-367 because CWE-367 is a general case time of check time of use weakness implemented as a DOS by causing a nonexistent file to be read where as this weakness is specifically a time of check time of use vulnerability that allows data leakage through arbitrary link following.

Weakness Class	Concurrency Handling
CWE	CWE-363
Variant	A
Language	Java
Status	WEAK-323 - CWE-363-01-Java DONE
Negative Technical Impact(s)	Read File

Implementation

The snippet takes two string as input, the first is the name of a control file that is used for delaying execution and the second is the name of the file to be read. The snippet first checks whether the file name of the file to be read contains any '/' characters, ensuring that it is in the current directory.

```
public static boolean isValidPath(String file) {
 Tracer.tracepointLocation("STONESOUP_INJECTED_FILE", "isValidPath");
    return !file.contains("/");
}
```

If the file is in the current directory, it is also checked to make sure that it is not a symbolic link.

```
public static boolean isSymlink(File file) throws IOException {
 Tracer.tracepointLocation("STONESOUP_INJECTED_FILE", "isSymlink");
    return !file.getCanonicalFile().equals(file.getAbsoluteFile());
}
```

Assuming that the file is both in the current directory and not a symbolic link, the snippet proceeds to create a file called <FILE_NAME>.pid, explained below in the inotifywait section, and read the control file. This will cause a delay in execution, and possibly allow the time of check time of use vulnerability to be exploited. This is the crossover point.

```
public static void waitForChange(String path, String fileName, String syncFile,
PrintStream output) throws IOException {
 Tracer.tracepointLocation("STONESOUP_INJECTED_FILE", "waitForChange");

    PrintWriter writer = new PrintWriter(path + fileName + ".pid");
    writer.close();

    Tracer.tracepointVariableString(".pid file", path + fileName + ".pid");

    Tracer.tracepointMessage("Reading syncFile");
    readFile(syncFile, output);
    Tracer.tracepointMessage("Finished reading syncFile");
}
```

If the control file delays the execution long enough for an external script to replace the input file, the snippet will open this unchecked file and print out the contents, possibly causing an information leak. This is the trigger point.

```
/* STONESOUP: TRIGGER POINT (Race Condition Enabling Link Following) */
stonesoup_reader = new BufferedReader(new
FileReader(stonesoup_file.getAbsolutePath()));
while ((stonesoup_line = stonesoup_reader.readLine()) != null) {
    stonesoup_output.println(stonesoup_line);
}
stonesoup_reader.close();
```

If the execution is not delayed long enough for an external script to replace the input file, the snippet will open the checked file and print the correct information, exiting without error.

External Scripts

The execution of this snippet in the bad cases is reliant on two external scripts which will be described in detail below.

runFifos.py

The runFifos.py script indirectly controls the time that the snippet takes between checking the validity of the input file and opening the file for reading. When run in the configuration required by this snippet, set by passing the script the --nosync flag, the script creates a FIFO file, writes 32 bytes of data to the file, and waits for it to be read. Once the first 32 bytes have been read, the script sleeps for 2 seconds and writes another 32 bytes to the file. After the second 32 bytes have been read the snippet closes the file and deletes it.

This script is able to control the amount of time between check and opening of the input file in the snippet due to the fact that FIFO files are blocking by default. The readFile call shown above will not return until the FIFO file is closed by the script, which has the file open for writing. This creates a robust way for influencing the execution time for the snippet.

The correct execution of this script for the snippet's bad test cases is shown below.

```
runFifos.py /opt/stonesoup/workspace/testData/fifo1 --nosync
```

Where /opt/stonesoup/workspace/testData/fifo1 is the name of the control file that will be passed to the snippet when it is executed.

The runFifos.py script must be executed as a backgrounded pre-process to the snippets execution, and allowed to continue during the execution of the snippet.

The source code for this script can be found here.

inotifywait

The inotifywait script is a one line bash script using the inotify-tools inotifywait command to check for the presence of a specific file created by the snippet, and upon finding that file, use it to identify the input file and replace it with a symbolic link. This script is essential to the proper functioning of the bad test cases.

The inotifywait script exists in the following form:

```
inotifywait -m -e create --format \"%f\" /opt/stonesoup/workspace/testData/ | while
read NEW_FILE; do
 if [[ \"${NEW_FILE}\" == *.pid ]]; then
  REPLACE_FILE=\"${NEW_FILE%.pid}\"; sleep 1; echo \"Replacing: \"
\"/opt/stonesoup/workspace/testData/${REPLACE_FILE}\" && \
  rm -f \"/opt/stonesoup/workspace/testData/${REPLACE_FILE}\" && ln -s /etc/passwd
\"/opt/stonesoup/workspace/testData/${REPLACE_FILE}\" && \
  echo \"Replaced.\";
 else
  echo \"Ignoring: \" \"/opt/stonesoup/workspace/testData/${NEW_FILE}\";
 fi;
done;
```

The script waits for the snippet to create a file in /opt/stonesoup/workspace/testData/ following the pattern <FILE_NAME>.pid where <FILE_NAME> is the name of the snippet's input file. Once the file has been detected the script sleeps for 1 second, deletes the snippets input file, and replaces it with a symbolic link to /etc/passwd.

The inotifywait script is run for each test case, good and bad, in order to more properly simulate the race condition present in the snippet. Because of this, the inotifywait utilizes a 1 second sleep before acting on the file in order to provide the snippet enough time to open and read the proper file in the good test cases, but still allow it to modify the file in the 2 second window allowed by the runFifos.py script in the bad test cases.

The inotifywait script must be executed as a backgrounded pre-process to the snippets execution and allowed to continue during the execution of the snippet.

Benign Inputs

Benign inputs all follow the pattern of a short text file as the control file and a valid text file in the current working directory as the input file.

The use of a short text file for the control file ensures that the snippets execution time between checking the validity of the input file and reading the file is negligible in relation to the timing of the inotifywait script.

Examples are as follows:

```
/opt/stonesoup/workspace/testData/file1 good_01.txt

Where /opt/stonesoup/workspace/testData/file1 is a text file containing no text and
good_01.txt is a text file in the current working directory containing the text
"FILE DATA GOES HERE YO".
```

```
/opt/stonesoup/workspace/testData/file2 good_02.txt

Where /opt/stonesoup/workspace/testData/file2 is a text file containing no text and
good_02.txt is a text file in the current working directory containing the text
"what does this file even mean?".
```

```
/opt/stonesoup/workspace/testData/file3 good_03.txt

Where /opt/stonesoup/workspace/testData/file3 is a text file containing no text and
good_03.txt is a text file in the current working directory containing the text
"Brah, ...bro, like, woah...".
```

Exploiting Inputs

Exploiting inputs all follow the pattern of a FIFO file controlled by runFifos.py as the control file, and a valid text file in the current working directory as the input file.

The use of a FIFO file controlled by runFifos.py ensures that the snippets execution time between checking the validity of the input file and reading the file is significant enough that the exploit is triggered and the snippet is tricked into leaking data by following a symbolic link.

For the sake of clarity, we assume that the FIFO file in each test case has been created by running the runFifos.py script in the background as follows:

```
runFifos.py <FIFO_FILE_NAME> --nosync
```

Where `<FIFO_FILE_NAME>` is the path to the FIFO file used as a control file in the snippet, for example: `/opt/stonesoup/workspace/testData/fifo1`.

Examples are as follows:

```
/opt/stonesoup/workspace/testData/fifo1 bad_0.txt

Where /opt/stonesoup/workspace/testData/fifo1 is a FIFO file controlled by
runFifos.py as shown above and bad_0.txt is a text file containing the text "HACK
THE PLANET! HACK THE PLANET!!!".
```

```
/opt/stonesoup/workspace/testData/fifo1 bad_1.txt

Where /opt/stonesoup/workspace/testData/fifo1 is a FIFO file controlled by
runFifos.py as shown above and bad_0.txt is a text file containing the text "1337
s<r1p7 15 f0r h4x0rz".
```

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Restrictions and Requirements

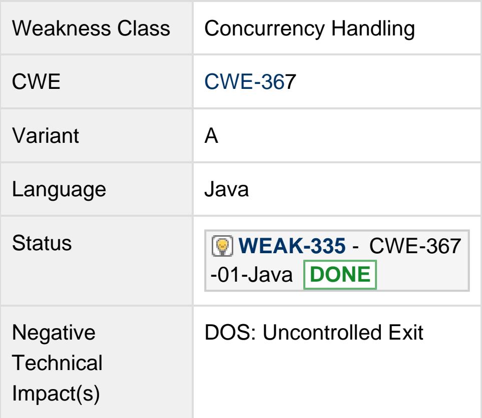
J - CWE-367A - Time-of-check Time-of-use (TOCTOU) Race Condition

Summary

This snippet implements a time of check time of use vulnerability that allows a DOS due to the input file being deleted before opening. The snippet takes in a control file and an input file. The input file is checked to see if it is in the current working directory and exists. If both of these conditions are true, the snippet opens and reads the control file and opens, reads, and prints the data in the input file. Since there is a delay between checking the validity of the file and opening it, the input file can be deleted before opening causing a DOS: Uncontrolled Exit. The control file is used as a method to delay execution long enough for this vulnerability to occur.

This is an example of a Multi-Process Race Condition.

This weakness differs from CWE-363 because CWE-363 is is specifically a time of check time of use vulnerability that allows data leakage through arbitrary link following, where as this weakness is a general case time of check time of use weakness implemented as a DOS by causing a nonexistent file to be read.



Weakness Class	Concurrency Handling
CWE	CWE-367
Variant	A
Language	Java
Status	WEAK-335 - CWE-367 -01-Java DONE
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

The snippet takes two string as input, the first is the name of a control file that is used for delaying execution and the second is the name of the file to be read. The snippet first checks whether the file name of the file to be read contains any '/' characters, ensuring that it is in the current directory.

```
public static boolean isValidPath(String file) {
    return !file.contains("/");
}
```

If the file is in the current directory, it is also checked to make sure that the file exists.

```
if (stonesoup_file.exists()) {
```

Assuming that the file is both in the current directory and exists, the snippet proceeds to create a file called <FILE_NAME>.pid, explained in the inotifywait section, and read the control file. This will cause a delay in execution, and possibly allow the time of check time of use vulnerability to be exploited. This is the crossover point.

```
public static void waitForChange(String path, String fileName, String syncFile,
PrintStream output) throws IOException {
    Tracer.tracepointLocation("STONESOUP_INJECTED_FILE", "waitForChange");

    PrintWriter writer = new PrintWriter(path + fileName + ".pid");
    writer.close();

    Tracer.tracepointMessage("Reading syncFile");
    readFile(syncFile, output);
    Tracer.tracepointMessage("Finished reading syncFile");
}
```

If the control file delays the execution long enough for an external script to delete the input file, the snippet will attempt to open and read a nonexistent file, causing the snippet to crash. This is the trigger point.

```
/* STONESOUP: TRIGGER POINT (time of check, time of use) */
stonesoup_reader = new BufferedReader(new
FileReader(stonesoup_file.getAbsolutePath()));
while ((stonesoup_line = stonesoup_reader.readLine()) != null) {
    stonesoup_output.println(stonesoup_line);
}
stonesoup_reader.close();
```

If the execution is not delayed long enough for an external script to delete the input file, the snippet will open the input file correctly and echo the contents to stdOut, exiting without error.

External Scripts

The execution of this snippet in the bad cases is reliant on two external scripts which will be described in detail below.

runFifos.py

The runFifos.py script indirectly controls the time that the snippet takes between checking the validity of the input file and opening the file for reading. When run in the configuration required by this snippet, set by passing the script the --nosync flag, the script creates a FIFO file, writes 32 bytes of data to the file, and waits for it to be read. Once the first 32 bytes have been read, the script sleeps for 2 seconds and writes another 32 bytes to the file. After the second 32 bytes have been read the snippet closes the file and deletes it.

This script is able to control the amount of time between check and opening of the input file in the snippet due to the fact that FIFO files are blocking by default. The readFile call shown above will not return until the FIFO file is closed by the script, which has the file open for writing. This creates a robust way for influencing the execution time for the snippet.

The correct execution of this script for the snippets bad test cases is shown below.

```
runFifos.py /opt/stonesoup/workspace/testData/fifo1 --nosync
```

Where `/opt/stonesoup/workspace/testData/fifo1` is the name of the control file that will be passed to the snippet when it is executed.

It is worth noting that the runFifos.py script must be executed as a backgrounded pre-process to the snippets execution, and allowed to continue during the execution of the snippet.

The source code for this script can be found here.

inotifywait

The inotifywait script is a one line bash script using the inotify-tools inotifywait command to check for the presence of a specific file created by the snippet, and upon finding that file, use it to identify the input file and replace it with a symbolic link. This script is essential to the proper

functioning of the bad test cases.

The inotifywait script exists in the following form:

```
inotifywait -m -e create --format \"%f\" /opt/stonesoup/workspace/testData/ | while
read NEW_FILE; do if [[ \"${NEW_FILE}\" == *.pid ]]; then
REPLACE_FILE=\"${NEW_FILE%.pid}\"; sleep 1; echo \"Deleting: \"
\"/opt/stonesoup/workspace/testData/${REPLACE_FILE}\" && rm -f
\"/opt/stonesoup/workspace/testData/${REPLACE_FILE}\" && echo \"Deleted.\"; else
echo \"Ignoring: \" \"/opt/stonesoup/workspace/testData/${NEW_FILE}\"; fi; done;
```

The script waits for the snippet to create a file in `/opt/stonesoup/workspace/testData/` following the pattern `<FILE_NAME>.pid` where `<FILE_NAME>` is the name of the snippet's input file. Once the file has been detected the script sleeps for 1 second and deletes the snippets input file.

The inotifywait script is run for each test case, good and bad, in order to more properly simulate the race condition present in the snippet. Because of this, the inotifywait utilizes a 1 second sleep before acting on the file in order to provide the snippet enough time to open and read the proper file in the good test cases, but still allow it to modify the file in the 2 second window allowed by the runFifos.py script in the bad test cases.

It is worth noting that the inotifywait script must be executed as a backgrounded pre-process to the snippets execution and allowed to continue during the execution of the snippet.

Benign Inputs

Benign inputs all follow the pattern of a short text file as the control file and a valid text file in the current working directory as the input file.

The use of a short text file for the control file ensures that the snippets execution time between checking the validity of the input file and reading the file is negligible in relation to the timing of the inotifywait script.

Examples are as follows:

```
/opt/stonesoup/workspace/testData/file1 good_0.txt
```

Where `/opt/stonesoup/workspace/testData/file1` is a text file containing no text and `good_0.txt` is a text file in the current working directory containing the text `"FILE DATA GOES HERE YO"`.

```
/opt/stonesoup/workspace/testData/file2 good_1.txt
```

Where `/opt/stonesoup/workspace/testData/file2` is a text file containing no text and good_1.txt is a text file in the current working directory containing the text `"what does this file even mean?"`.

```
/opt/stonesoup/workspace/testData/file3 good_2.txt
```

Where `/opt/stonesoup/workspace/testData/file3` is a text file containing no text and `good_2.txt` is a text file in the current working directory containing the text `"Brah, ...bro, like, woah..."`.

Exploiting Inputs

Exploiting inputs all follow the pattern of a FIFO file controlled by runFifos.py as the control file, and a valid text file in the current working directory as the input file.

The use of a FIFO file controlled by runFifos.py ensures that the snippets execution time between checking the validity of the input file and

reading the file is significant enough that the exploit is triggered and the snippet crashes by attempting to open and read a nonexistent file.

For the sake of clarity, we assume that the FIFO file in each test case has been created by running the runFifos.py script in the background as follows:

```
runFifos.py <FIFO_FILE_NAME> --nosync
```

Where `<FIFO_FILE_NAME>` is the path to the FIFO file used as a control file in the snippet, for example: `/opt/stonesoup/workspace/testData/fifo1`.

Examples are as follows:

```
/opt/stonesoup/workspace/testData/fifo1 bad_0.txt
```

Where `/opt/stonesoup/workspace/testData/fifo1` is a FIFO file controlled by `runFifos.py` as shown above and `bad_0.txt` is a text file containing the text `"HACK THE PLANET! HACK THE PLANET!!!"`.

```
/opt/stonesoup/workspace/testData/fifo2 bad_1.txt
```

Where `/opt/stonesoup/workspace/testData/fifo2` is a FIFO file controlled by `runFifos.py` as shown above and `bad_0.txt` is a text file containing the text `"1337 s<r1p7 15 f0r h4x0rz"`.

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Restrictions and Requirements

J - CWE-412A - Unrestricted Externally Accessible Lock

Summary

This snippet takes the name of an externally accessible file as input, and treats the file as a global mutex lock. The snippet will attempt to "grab" the mutex lock by checking for the files existence, and creating it if it doesn't exist. The creation of the file is treated as grabbing the lock, and if the snippet is able to do so it will enter the critical section and return, deleting the file to "release" the lock, and finishing execution without error. However, if the file already exists, the snippet will continually poll the existence of the file, waiting to be able to "grab" the lock by creating the file, resulting in a paused state for as long as the file exists.

Weakness Class	Concurrency Handling
CWE	CWE-412
Variant	A
Language	Java
Status	WEAK-339 - CWE-412-01-Java DONE

Negative Technical Impact(s)	DOS: Infinite Loop

Implementation

This snippet takes in the name of an externally accessible file as input and creates a file object with that path. This is the crossover point.

```
/* STONESOUP: CROSS-OVER POINT (Unrestricted Externally Accessible Lock) */
stonesoup_file = new File(stonesoup_path, stonesoup_value);
```

The snippet then attempts create the file if it doesn't exists, looping until it is able to do so. If the file already exists, this will result in a paused state for as long as the file exists. This is the trigger point.

```
/* STONESOUP: TRIGGER POINT (Unrestricted Externally Accessible Lock) */
stonesoup_output.println(stonesoup_path);
Tracer.tracepointMessage("Attempting to grab file lock.");
while (!stonesoup_file.createNewFile()) {
    Thread.sleep(1);
}
```

If the file does not exists, the snippet will create the file, print to the output stream that it has grabbed the lock, and finally delete the file again, completing execution without error.

Benign Inputs

All valid names of nonexistent files that are writable by the snippet are benign. For example:

```
file.txt
```

Were `file.txt` does not exist.

```
new.txt
```

Where `new.txt` does not exist.

```
program.cfg
```

Where `program.cfg` does not exist.

Exploiting Inputs

All valid names of currently existing files that are writable by the snippet are exploitative. For example:

```
file_exists.txt
```

Where `file_exists.txt` is a text file containing the text `"data1 data2 data3 data4 data5 data6 data7"`.

```
notnew.txt
```

Where `notnew.txt` is a text file containing the text `"data1 data2 data3 data4 data5 data6 data7"`.

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Restrictions and Requirements

J - CWE-414A - Missing Lock Check

Summary

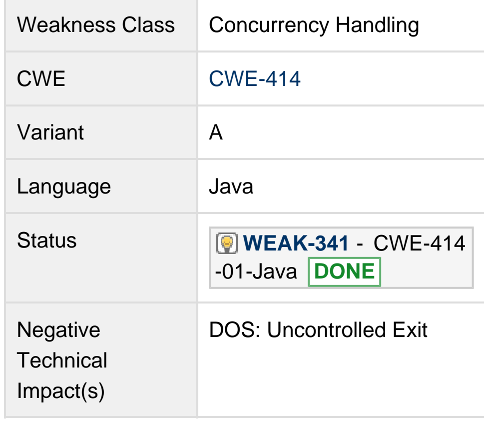
This snippet implements a missing lock check that allows two threads to access a shared object simultaneously, leading to a null pointer deference. It takes an integer, the names of two control files, and an input string as input. The integer and two control files are used for timing, and are discussed in more detail below. The snippet creates two threads that both modify a shared object containing the input string, however only one of the the threads implements a locking mechanism. Since one thread sets the object to null temporarily and the other tries to read the reference, this can cause a null pointer dereference if the timing occurs properly.

This is an example of a Multi-Threaded Race Condition.

This snippet differs from the snippet for CWE-567 because this snippet implements two threads that access a shared object but only one of them implements a locking mechanism and the snippet for CWE-567 spawns two thread and neither thread implements a locking mechanism while both access a static shared variable.

This snippet differs from the snippet for CWE-820 because the snippet for CWE-820 spawns two threads and neither thread implements a locking mechanism while both access a shared object.

This snippet differs from the snippet for CWE-821 because the snippet for CWE-821 spawns two threads which each use a separate lock object.



Weakness Class	Concurrency Handling
CWE	CWE-414
Variant	A
Language	Java
Status	WEAK-341 - CWE-414-01-Java DONE
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

This snippet contains two main static classes: toCap and delNonAlpha, both subclasses of Runnable, and run in a thread pool. Both classes act upon a shared string (copied from the input), however, only toCap implements a mutex lock before accessing the shared memory. The threads executing both classes are spawned at the same time, and execute concurrently. The execution of each class is explained below.

delNonAlpha

The delNonAlpha thread is called with the name of a control file, and the stonesoup output stream. Once executing, this thread creates a temporary string and iterates through the shared string, copying all alpha characters into the new string. Once the temp string contains all of the alpha characters from the shared string, the snippet enters the crossover point by setting the shared string to NULL. The control file is then read, allowing our test case to influence the execution timing of the thread (explained in the next section).

```
for (int i = 0; i < stonesoup_threadInput.length(); i++) {
    if(Character.isLetter(stonesoup_threadInput.charAt(i))) {
        temp.append(stonesoup_threadInput.charAt(i));
    }
}

Tracer.tracepointMessage("CROSSOVER-POINT: BEFORE");
/* STONESOUP: CROSSOVER POINT (missing lock check) */
stonesoup_threadInput = null;
Tracer.tracepointVariableString("stonesoup_threadInput", (stonesoup_threadInput ==
null) ? "(null)" : stonesoup_threadInput.toString());
Tracer.tracepointMessage("CROSSOVER-POINT: AFTER");

readFile(filename, output);
```

Once the control file is finished reading, assuming the bad case has not been executed causing a null pointer dereference (explained below), the thread sets the shared string equal to the newly filled temp string.

toCap

The toCap thread is called with the input integer, name of a control file, and the stonesoup output stream. Once executing, this thread allocates an integer array with size equal to the input integer, locks the mutex lock, and populates the newly created array with integers in descending order (from size to 0). This array is then sorted to ascending order. The purpose of the sort is to slow down the execution of the snippet in certain good test cases (explained in more detail in the next section).

Once the array sort is finished, the thread reads its control file (again used for delaying the snippets execution, explained below), and enters the trigger point by accessing the shared string and converting it to upper case. If the other thread has set the shared string to NULL but not yet reassigned it a value the string access will result in a null pointer dereference exception being thrown.

```
int [] sortMe = new int[size];

lock.lock();

for (int i = 0; i < size; i++) {
    sortMe[i] = size - i;
}
Arrays.sort(sortMe);
readFile(filename, output);

Tracer.tracepointMessage("TRIGGER-POINT: BEFORE");
/* STONESOUP: TRIGGER POINT (missing lock check) */
stonesoup_threadInput = new
StringBuilder().append(stonesoup_threadInput.toString().toUpperCase());
Tracer.tracepointMessage("TRIGGER-POINT: AFTER");

lock.unlock();
```

If the other snippet has finished execution and reset the shared string to a valid string object, the string access will occur normally, the thread will unlock it's mutex, and the snippet will finish execution without error.

Timing Explanation

Due to the random nature of race conditions, we cannot rely on the thread scheduler alone to run our threads and expect that they will deterministically result in either a good or bad outcome. In order to provide a deterministic execution environment we introduced two control mechanisms into the threads themselves. The first is a set of two control files which, when created as FIFO files and controlled by the runFifos.py script described below, ensure that one file read completes two seconds before the other. The other control method is to pass in two regular files for the control files, essentially negating the readFile() commands, and instead passing a significantly large integer in order to influence the thread timing though an array sort on an array of descending integers with size equal to the input integer.

For more information on either of these approaches, view the documentation for multi-threaded race conditions found here.

runFifos.py

The runFifos.py script indirectly controls the time execution order of the threads in a snippet, the specific configuration described below is set by passing two file names of nonexistent files to the runFifos.py script. The script will create both files and control them to ensure that the first FIFO closes two seconds before the second, allowing them to be used to control the execution of the snippet. This is done by writing 32 bytes of data to both FIFOs and waiting for them both to be read, ensuring that both threads in the snippet are synchronized on the file read. The script then writes a second 32 bytes to the first FIFO file and closes it. It then waits two seconds and writes a second 32 bytes to the second FIFO file and closes it as well. The order is as follows:

```
t1: wrote first 32 bytes to file
t2: wrote first 32 bytes to file
ALL: threads synchronized
t1: wrote second 32 bytes to file
t2: t1 finished
t2: wrote second 32 bytes to file
```

The correct execution of this script for the snippets bad test cases is shown below.

```
runFifos.py /opt/stonesoup/workspace/testData/fifo1
/opt/stonesoup/workspace/testData/fifo2
```

Where `/opt/stonesoup/workspace/testData/fifo1` and `/opt/stonesoup/workspace/testData/fifo2` are the names of the control files that will be passed to the snippet when it is executed.

It is worth noting that the runFifos.py script must be executed as a backgrounded pre-process to the snippets execution and allowed to continue to run during the execution of the snippet.

The source code for this script can be found here.

Benign Inputs

Benign inputs come in two forms. The first is using FIFO files with the order passed to the snippet reversed from the order called in the `runFifos.py` script. When executing benign test cases in this format the value of the input integer does not change the execution of the snippet and only serves to slow the overall execution of the snippet when the value is sufficiently large. The second format is to call the snippet with two regular files as control files and a significantly large control integer.

For the following examples that use FIFO files we will assume that the `runFifos.py` script is executed in the format show above.

FIFO Files

Examples of benign inputs with FIFO control files are as follows:

```
50 /opt/stonesoup/workspace/testData/fifo2 /opt/stonesoup/workspace/testData/fifo1
thisisFine
```

```
50 /opt/stonesoup/workspace/testData/fifo2 /opt/stonesoup/workspace/testData/fifo1
laLAla
```

Regular Files

Examples of benign inputs with regular control files are as follows:

```
5000000 /opt/stonesoup/workspace/testData/file1
/opt/stonesoup/workspace/testData/file2 asdfdsa
```

Exploiting Inputs

Exploring inputs all use FIFO control files passed in the same order as they are called in the `runFifos.py` script. Like the benign test case using FIFO control files above, the value of the integer input is irrelevant and simply serves to slow the overall execution of the snippet when the value is sufficiently large. For example:

```
50 /opt/stonesoup/workspace/testData/fifo1 /opt/stonesoup/workspace/testData/fifo2
111
```

```
50 /opt/stonesoup/workspace/testData/fifo1 /opt/stonesoup/workspace/testData/fifo2
What_a_lovely_day_it_is.
```

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Restrictions and Requirements

J - CWE-543A - Use of Singleton Pattern Without Synchronization in a Multithreaded Context

Summary

This snippet implements a singleton class without synchronization that can lead to two threads instantiating separate instances of the singleton resulting in a deadlocked state. It takes a control integer, the names of two control files, and another integer as input. The control integer and the two control files are used for timing within the snippet to ensure that we hit either a good or bad case, and the input integer is used as the number of Fibonacci values to calculate. The snippet then spawns two threads, one to calculate Fibonacci values and push them to a queue, and one to read the values from the queue and print them to the output stream. The shared queue is implemented as a singleton and is a blocking queue, if both threads create a separate instance of the singleton they will block indefinitely causing a deadlock.

This is an example of a Multi-Threaded Race Condition.

Weakness Class	Concurrency Handling
CWE	CWE-543
Variant	A
Language	Java
Status	WEAK-347 - CWE-543 -01-Java DONE
Negative Technical Impact(s)	DOS: Deadlock

Implementation

This snippet contains three main static classes: logData, printData, and JobHandler, the first two are subclasses of Runnable, and run in a thread pool, and the third is a singleton class used to share data between the threads. Both threaded classes make a call to the instantiation method of the JobHandler, and assuming that the singleton performed as expected, calculate n Fibonacci numbers and push them to the JobHandler queue, where n is the input integer, and grab the data from the JobHandler queue and print them to the output stream, respectively.

logData

The logData thread is called with the input integer, control integer, name of a control file, and the stonesoup output stream. Once called, the thread allocates an integer array with size equal to the control integer, and populates the newly created array with integers in descending order (from size to 0). This array is then sorted to ascending order. The purpose of the sort is to slow down the execution of the snippet in certain good test cases (explained in more detail in the next section). Once the array sort is finished, the thread reads it's control file (again to control the execution time of the thread, explained below), and gets an instance of the shared blocking queue.

```
int [] sortMe = new int[size];

for (int i = 0; i < size; i++) {                         // Slow things down for
timing
    sortMe[i] = size - i;
}
Arrays.sort(sortMe);

readFile(filename, output);

JobHandler jobs = JobHandler.getInstance(filename, output);
```

The thread then starts calculating Fibonacci values from 0 to the input integer, pushing them to the blocking queue. If the blocking queue is never emptied, and the thread fills it's buffer, it will block indefinitely causing this thread to hang.

```
// populate data
BigInteger a1 = BigInteger.valueOf(0);                       // Calculate Fibonacci
Sequence
BigInteger a2 = BigInteger.valueOf(0);
BigInteger c = BigInteger.valueOf(0);
for (int i = 0; i < numVal; i++) {
    if (i == 0) {
        jobs.enqueue(BigInteger.valueOf(0));
    }
    else if (i == 1) {
        a1 = BigInteger.valueOf(1);
        jobs.enqueue(BigInteger.valueOf(0));
    }
    else {
        c = a1.add(a2);
        a2 = a1;
        a1 = c;
        jobs.enqueue(c);
    }
}
jobs.enqueue(BigInteger.valueOf(-1));
```

If the queue is being emptied, the thread will successfully calculate the Fibonacci numbers and exit without error.

printData

The printData thread is called with the name of a control file and the stonesoup output stream. Once called, the thread grabs an instance of the shared blocking queue, and starts reading the data from the queue, printing it to the output stream. If the thread gets a unique instance of the queue, rather than the intended singleton instance, the queue will never have data to read and the thread will block indefinitely, causing this thread to hang. This is the trigger point.

```
JobHandler jobs = JobHandler.getInstance(filename, output);
BigInteger i;

Tracer.tracepointBuffer("printData: UID of JobHandler",
Integer.toHexString(System.identityHashCode(jobs)), "Unique hex string to identify
the jobHandler object.");
Tracer.tracepointMessage("TRIGGER-POINT: BEFORE");
/* STONESOUP: TRIGGER POINT (Singleton without Sync) */
while ((i = jobs.dequeue()) != BigInteger.valueOf(-1)) {     // Read from
BlockingQueue, will wait indefinitely
    output.println(i.toString(10));                          //    if queue is never
populated.
}
```

If this thread gets a proper instance of the shared blocking queue, it will print all of the calculated Fibonacci values to the output stream and exit without error.

JobHandler

The JobHandler class is implemented as a singleton class but does not implement locking, allowing a race condition where if one thread requests an instance while another thread is in the getInstance method, both threads can get a unique instance. In order to ensure that this race condition happens during a bad test case and prevent it from happening in a good, the getInstance method and the logData classes have a call to readFile() using the two control files. The call to initialize() is the crossover point.

When the snippet is called with two FIFO files as control files, the first FIFO file gets passed to logData and the second to printData which in turn passes that on to the JobHandler's initialize() function shown below. When the FIFO files are passed in the same order as they are

passed to the runFifos.py script logData is allowed to call JobHandler's initialize() function while printData is waiting at the readFile() call in the initialize() function. This will cause both printData and logData to initialize unique instances of JobHandler leading to a deadlock. If the files are passed in revers order from the call to runFifos.py the opposite happens: logData hangs before calling initialize() until after printData has finished initialization of the JobHandler and they both return with the same instance of the class.

```
if (instance == null) {
    Tracer.tracepointMessage("CROSSOVER-POINT: BEFORE");
    /* STONESOUP: CROSS-OVER POINT (Singleton without Sync) */
    readFile(filename, output);

    JobHandler temp = new JobHandler();
    temp.initialize();
    instance = temp;
    Tracer.tracepointMessage("CROSSOVER-POINT: AFTER");
    return temp;
}
return instance;
```

If the snippet is called with two regular files as control files the timing is entirely dependent on the control integer. This integer is used to create, populate, and and sort an array in order to slow down the execution of the logData thread to allow the printData thread to finish initialization of the JobHandler singleton before logData is allowed to continue and grab the instance. In order to ensure that this execution sequence occurs, the integer must be sufficiently large. In the case where FIFO files are used for timing, the control integer is irrelevant, however a significantly large number will still slow down the overall execution of the snippet.

Timing Explanation

Due to the random nature of race conditions, we cannot rely on the thread scheduler alone to run our threads and expect that they will deterministically result in either a good or bad outcome. In order to provide a deterministic execution environment we introduced two control mechanisms into the threads themselves. The first is a set of two control files which, when created as FIFO files and controlled by the runFifos.py script described below, ensure that one file read completes two seconds before the other, the order of which is described in the next section. The other control method is to pass in two regular files for the control files, essentially negating the readFile() commands, and instead passing a significantly large integer in order to influence the thread timing though an array sort on an array of descending integers with size equal to the input integer.

For more information on either of these approaches, view the documentation for multi-threaded race conditions found here, and read the section above about the JobHandler class which goes into more detail about how these control mechanisms work in this particular snippet.

runFifos.py

The runFifos.py script indirectly controls the time execution order of the threads in a snippet, the specific configuration described below is set by passing two file names of nonexistent files to the runFifos.py script. The script will create both files, and control them to ensure that the first FIFO closes two seconds before the second, allowing them to be used to control the execution of the snippet. This is done by writing 32 bytes of data to both FIFOs and waiting for them both to be read, ensuring that both threads in the snippet are synchronized on the file read. The script then writes a second 32 bytes to the first FIFO file, and closes it. It then waits two seconds and writes a second 32 bytes to the second FIFO file and closes it as well. The order is as follows:

```
t1: wrote first 32 bytes to file
t2: wrote first 32 bytes to file
ALL: threads synchronized
t1: wrote second 32 bytes to file
t2: t1 finished
t2: wrote second 32 bytes to file
```

The correct execution of this script for the snippets bad test cases is shown below.

```
runFifos.py /opt/stonesoup/workspace/testData/fifo1
/opt/stonesoup/workspace/testData/fifo2
```

Approved for public release; distribution unlimited.

Where `/opt/stonesoup/workspace/testData/fifo1` and `/opt/stonesoup/workspace/testData/fifo2` are the names of the control files that will be passed to the snippet when it is executed.

It is worth noting that the runFifos.py script must be executed as a backgrounded pre-process to the snippets execution, and allowed to continue during the execution of the snippet.

The source code for this script can be found here.

Benign Inputs

Benign inputs come in two forms. The first is using FIFO files with the order passed to the snippet reversed from the order called in the `runFifos.py` script. When executing benign test cases in this format the value of the input integer does not change the execution of the snippet, and only serves to slow the overall execution of the snippet when the value is sufficiently large. The second format is to call the snippet with two regular files as control files and a significantly large input integer, used to control the execution of the threads.

For the following examples that use FIFO files we will assume that the `runFifos.py` script is executed in the format show above.

FIFO Files

Examples of benign inputs with FIFO control files are as follows:

```
42 /opt/stonesoup/workspace/testData/fifo2 /opt/stonesoup/workspace/testData/fifo1
42
```

```
670000 /opt/stonesoup/workspace/testData/fifo2
/opt/stonesoup/workspace/testData/fifo1 50
```

Regular Files

Examples of benign inputs with regular control files are as follows:

```
5000000 /opt/stonesoup/workspace/testData/file1
/opt/stonesoup/workspace/testData/file2 50
```

Exploiting Inputs

Exploring inputs all use FIFO control files passed in the same order as they are called in the `runFifos.py` script. Like the benign test case using FIFO control files above, the value of the integer input is irrelevant and simply serves to slow the overall execution of the snippet when the value is sufficiently large. For example:

```
500000 /opt/stonesoup/workspace/testData/fifo1
/opt/stonesoup/workspace/testData/fifo2 50
```

```
50 /opt/stonesoup/workspace/testData/fifo1 /opt/stonesoup/workspace/testData/fifo2
500
```

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Restrictions and Requirements

J - CWE-567A - Unsynchronized Access to Shared Data in a Multithreaded Context

Summary

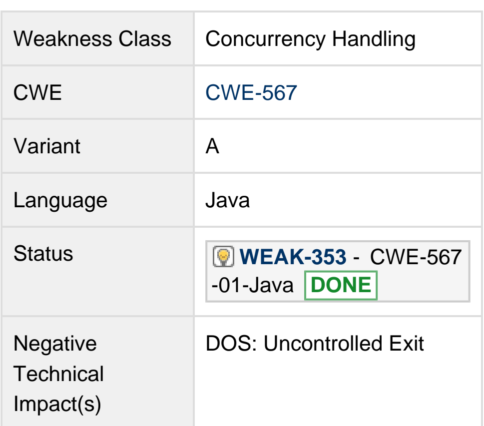
This snippet implements two threads that access a shared object without synchronization. The snippet takes a control integer, the names of two control files, and an input string. The control integer and the two control files are used for timing within the snippet to ensure that we hit either a good or bad case, and the input string is used as shared data for the threads to act upon. When executing, one thread calculates a divisor by subtracting 'A' from the first character of the input string, and, if the devisor is less than or equal to zero, sets the devisor to a positive number. The other thread uses this divisor to calculate a new string by dividing each character by the divisor. If the input string starts with an 'A' and the second thread attempts to calculate the new string before the divisor has been sanitized, a divide by zero error will occur.

This is an example of a Multi-Threaded Race Condition.

This snippet differs from the snippet for CWE-414 because this snippet spawns two threads, neither of which implement a lock check, which both access a shared static variable where as the snippet for CWE-414 spawns two threads and only one thread implements a locking mechanism while accessing a shared resource.

This snippet differs from the snippet for CWE-820 because the snippet for CWE-820 spawns two threads and neither thread implements a locking mechanism while both access a shared object. It should be noted that this snippet is very similar to the snippet for CWE-820 because CWE-567 is a child of CWE-820. The primary difference between the two snippets resides in the use of a shared static variable in the case of this snippet and a shared object in the case of the snippet for CWE-820.

This snippet differs from the snippet for CWE-821 because the snippet for CWE-821 spawns two threads which each use a separate lock object.



Weakness Class	Concurrency Handling
CWE	CWE-567
Variant	A
Language	Java
Status	WEAK-353 - CWE-567-01-Java DONE
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

This snippet contains two main static classes: calcDevAmount, and devChar, both of which are subclasses of Runnable, and run in a thread pool. The two classes are described below.

calcDevAmount

The calcDevAmount thread is called with the name of the second control file and the stonesoup output stream. The thread then calculates a devisor by subtracting 'A' from the first character of the input string. Once the devisor is calculated, the snippet reads the control file (used to control the execution time of the thread, explained below), pausing for long enough for the race condition to be executed in a bad test case. This is the crossover point.

```
/* STONESOUP: CROSS-OVER POINT (unsynchronized access to shared data) */
dev_amount = stonesoup_threadInput.charAt(0) - 'A';

readFile(filename, output);
```

If the input string did not start with an 'A' or the control files/integer prevented the race condition, the snippet proceeds to sanitize the input.

```
if (dev_amount < 0) {
    dev_amount *= -1;
}
if (dev_amount == 0) {
    dev_amount += 1;
}
```

At this point the thread has completed execution and will join without error.

devChar

The devChar thread is called with the name of the first control file, the control integer, and the stonesoup output stream. The thread then allocates an integer array with size equal to the control integer, and populates the newly created array with integers in descending order (from size to 0). This array is then sorted to ascending order. The purpose of the sort is to slow down the execution of the snippet in certain good test cases (explained in more detail in the next section). Once the array sort is finished, the thread reads it's control file (again to control the execution time of the thread, explained below), and calculates a new string by dividing each character in the input string by the divisor calculated in the other thread. If the divisor is zero, a divide by zero error occurs. This is the trigger point.

```
for (int i = 0; i < size; i++) {
    sortMe[i] = size - i;
}
Arrays.sort(sortMe);

readFile(filename, output);

Tracer.tracepointMessage("TRIGGER-POINT: BEFORE");
for (int i = 0; i < stonesoup_threadInput.length(); i++) {
    /* STONESOUP: TRIGGER POINT (unsynchronized access to shared data) */
    stonesoup_threadInput.setCharAt(i, (char)(stonesoup_threadInput.charAt(i) /
dev_amount));
}
```

If the divisor is non-zero the thread will complete execution without error and join back with the main thread.

Timing Explanation

Due to the random nature of race conditions, we cannot rely on the thread scheduler alone to run our threads and expect that they will deterministically result in either a good or bad outcome. In order to provide a deterministic execution environment we introduced two control mechanisms into the threads themselves. The first is a set of two control files which, when created as FIFO files and controlled by the runFifos.py script described below, ensure that one file read completes two seconds before the other. The other control method is to pass in two regular files for the control files, essentially negating the readFile() commands, and instead passing a significantly large integer in order to influence the thread timing though an array sort on an array of descending integers with size equal to the input integer.

For more information on either of these approaches, view the documentation for multi-threaded race conditions found here.

runFifos.py

The runFifos.py script indirectly controls the time execution order of the threads in a snippet, the specific configuration described below is set by passing two file names of nonexistent files to the runFifos.py script. The script will create both files and control them to ensure that the first FIFO closes two seconds before the second, allowing them to be used to control the execution of the snippet. This is done by writing 32 bytes of data to both FIFOs and waiting for them both to be read, ensuring that both threads in the snippet are synchronized on the file read. The script then writes a second 32 bytes to the first FIFO file and closes it. It then waits two seconds and writes a second 32 bytes to the second FIFO file and closes it as well. The order is as follows:

```
t1: wrote first 32 bytes to file
t2: wrote first 32 bytes to file
ALL: threads synchronized
t1: wrote second 32 bytes to file
t2: t1 finished
t2: wrote second 32 bytes to file
```

The correct execution of this script for the snippets bad test cases is shown below.

```
runFifos.py /opt/stonesoup/workspace/testData/fifo1
/opt/stonesoup/workspace/testData/fifo2
```

Where `/opt/stonesoup/workspace/testData/fifo1` and `/opt/stonesoup/workspace/testData/fifo2` are the names of the control files that will be passed to the snippet when it is executed.

It is worth noting that the runFifos.py script must be executed as a backgrounded pre-process to the snippets execution and allowed to continue to run during the execution of the snippet.

The source code for this script can be found here.

Benign Inputs

Benign inputs come in two forms. The first is using FIFO files with the order passed to the snippet reversed from the order called in the `runFifos.py` script. When executing benign test cases in this format the value of the input integer does not change the execution of the snippet and only serves to slow the overall execution of the snippet when the value is sufficiently large. The second format is to call the snippet with two regular files as control files and a significantly large control integer.

For the following examples that use FIFO files we will assume that the `runFifos.py` script is executed in the format show above.

FIFO Files

Examples of benign inputs with FIFO control files are as follows:

```
50 /opt/stonesoup/workspace/testData/fifo2 /opt/stonesoup/workspace/testData/fifo1
abcdabcd
```

```
50 /opt/stonesoup/workspace/testData/fifo2 /opt/stonesoup/workspace/testData/fifo1
somevalue
```

Regular Files

Examples of benign inputs with regular control files are as follows:

```
5000000 /opt/stonesoup/workspace/testData/file1
/opt/stonesoup/workspace/testData/file2 Astringvalue
```

Exploiting Inputs

Exploring inputs all use FIFO control files passed in the same order as they are called in the `runFifos.py` script. Like the benign test case using FIFO control files above, the value of the integer input is irrelevant and simply serves to slow the overall execution of the snippet when the value is sufficiently large. For example:

```
50 /opt/stonesoup/workspace/testData/fifo1 /opt/stonesoup/workspace/testData/fifo2
ABC123
```

```
50 /opt/stonesoup/workspace/testData/fifo1 /opt/stonesoup/workspace/testData/fifo2
Anumb3rYO
```

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Restrictions and Requirements

J - CWE-572A - Call to Thread run() instead of start()

Summary

This snippet implements an incorrect call to the run() method of a runnable object instead of calling start(). The snippet takes in the name of an input file which contains data. The first 4 bytes of the file are interpreted as a 32-bit integer describing the length of the file, and that number of bytes is read from the file. If the size is greater than the actual length of the file a RuntimeException is raised. Since the thread was called improperly by calling it's run() method the exception will be passed to the main thread and handled improperly.

Weakness Class	Concurrency Handling
CWE	CWE-572
Variant	A
Language	Java
Status	**WEAK-357** - CWE-572-01-Java DONE
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

This snippet contains one main class, HelloRunnable which is a subclass of Runnable and is intended to be run in a thread pool but is instead run by calling it's run() method. This is both the crossover point and the trigger point.

```
/* STONESOUP: CROSS-OVER POINT (Call to Thread run instead of start) */
/* STONESOUP: TRIGGER POINT (Call to Thread run instead of start) */
stonesoup_thread1.run();
```

Once called, the HelloRunnable class checks to see that the input file both exists and is a file, and proceeds to open the file and read it to a byte array.

```
byte[] inputBuffer = new byte[1024];
ByteArrayOutputStream byteArrayOutputStream = new ByteArrayOutputStream();

int readAmount = 0;
while ((readAmount = inputStream.read(inputBuffer)) != -1) {
    Tracer.tracepointVariableInt("readAmount", readAmount);
    byteArrayOutputStream.write(inputBuffer, 0, readAmount);
}

data = byteArrayOutputStream.toByteArray();
```

Once the data is read to the byte array, the snippet checks to make sure there is at least 4 bytes to ensure that the length integer is valid. The snippet then converts the first 4 bytes to a 32-bit integer, uses this integer as the number of bytes to read, and attempts to read in that number of bytes.

```
ByteBuffer buffer = ByteBuffer.wrap(data);
int dataLength = buffer.getInt();

/* ensure that data length is positive to avoid invalid reads */
if (dataLength < 0) {
    return;
} else if (dataLength == 0) {
    Tracer.tracepointError("Received payload with no data.");
    this.output.println("Received payload with no data.");
    return;
}


/* read the payload of the indictaed size */
byte[] payload = new byte[dataLength];
Tracer.tracepointBufferInfo("payload", payload.length, "Length of they payload byte
array.");
Tracer.tracepointBufferInfo("buffer.position", buffer.position(), "Position in
buffer.");
buffer.get(payload);
```

If the integer indicated that there were more bytes than actually exist a RuntimeException will be thrown to the main thread. This should never happen if a thread is called properly.

Benign Inputs

Benign inputs all contain the name of a binary file who's first 4 bytes represent a 32-bit integer containing the length of the rest of the file. For example:

```
file.bin
```

Where `file.bin` is a file containing the 32-bit integer 10, followed by 10 'A's.

```
new.bin
```

Where `new.bin` is a file containing the 32-bit integer 24, followed by the text "This message seems valid".

```
program.bin
```

Where `program.bin` is a file containing the 32-bit integer 636, followed by the text "Wood exhibits its greatest strength in tension parallel to the grain, and it is very uncommon in practice for a specimen to be pulled in two lengthwise. This is due to the difficulty of making the end fastenings secure enough for the full tensile strength to be brought into play before the fastenings shear off longitudinally. This is not the case with metals, and as a result they are used in almost all places where tensile strength is particularly needed, even though the remainder of the structure, such as sills, beams, joists, posts, and flooring, may be of wood. Thus in a wooden truss bridge the tension members are steel rods.".

Exploiting Inputs

Exploiting inputs all contain the name of a binary file who's first 4 bytes represent a 32-bit integer containing a number that is greater than the length of the rest of the file. For example:

```
message.bin
```

Where `message.bin` is a file containing the 32-bit integer 7, followed by the text "POTATO".

```
message.bin
```

Where `message.bin` is a file containing the 32-bit integer 2048, followed by the text "HAT".

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Restrictions and Requirements

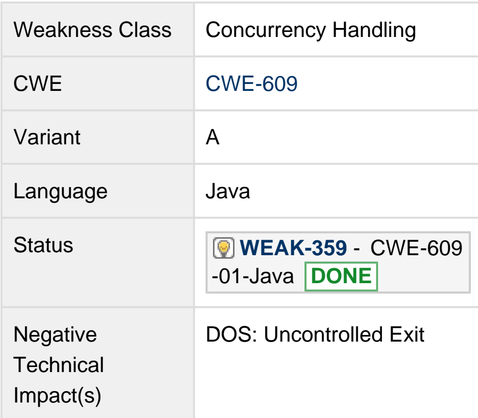
J - CWE-609A - Double-Checked Locking

Summary

This snippet implements a double checked lock around the initialization of a shared static data class in an attempt to be efficient. The snippet takes a control integer, the names of two control files, and an input string. The control integer and the two control files are used for timing within the snippet to ensure that we hit either a good or bad case, and the input string is used as shared data for the threads to act upon.

When executing, the snippet spawns two threads which in turn attempt to get a shared instance of the input string by calling a double-checked locking function that initializes and/or returns a shared instance of the input string. If one thread calls this function while the other one is inside the function initializing the string it can return an uninitialized copy of the instance to the second thread. This will lead to an access of uninitialized data resulting in a StringIndexOutOfBoundsException.

This is an example of a Multi-Threaded Race Condition.



Weakness Class	Concurrency Handling
CWE	CWE-609
Variant	A
Language	Java
Status	WEAK-359 - CWE-609-01-Java DONE
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

This snippet contains two main static classes and a static function: doStuff, doStuff2, and init_Stonesoup_str, respectively. Both classes are subclasses of Runnable and are run in a thread pool, and the function is used to implement the weakness. The two classes and the function are described below.

doStuff

The doStuff thread is called with the input string, the control integer, the second control file, and the stonesoup output stream. Once executing, the thread calls init_Stonesoup_Str(), passing its input string, control integer, control file, and output stream.

```
init_Stonesoup_Str(data, size, filename, output);
```

Once init_Stonesoup_Str() returns the thread prints out the data of the newly initialized Stonesoup_Str and returns.

```
output.println(Thread.currentThread().getId() + ": In doStuff Stonesoup_Str is: " +
Stonesoup_Str.data.toString());
```

The crossover point resides in the call to init_Stonesoup_Str() and will be described below.

doStuff2

The doStuff2 thread is called with the input string, the control integer, the first control file, and the stonesoup output stream. Once executing, the thread allocates an integer array with size equal to the control integer, and populates the newly created array with integers in descending order (from size to 0). This array is then sorted to ascending order. The purpose of the sort is to slow down the execution of the snippet in certain good test cases (explained in more detail in the next section). Once the array sort is finished, the thread reads it's control file (again to control the execution time of the thread, explained below), and calls init_Stonesoup_Str() with its input string, control integer, and output stream, passing a null value instead of a control file.

```
int[] sortMe = new int[size];

try {
    output.println(Thread.currentThread().getId() + ": Inside doStuff2");
    for (int i = 0; i < size; i++){
        sortMe[i] = size - i;
    }
    Arrays.sort(sortMe);

    readFile(filename, output);
    Tracer.tracepointMessage("doStuff2: entering init_Stonesoup_Str");
    init_Stonesoup_Str(data, size, null, output);
```

Once the call to init_Stonesoup_Str() returns, the snippet safely iterates through the newly initialized Stonesoup_Str and subtracts 32 from each character. The thread then attempts to access the first element of the newly initialized string, and if this thread called init_Stonesoup_Str() while the other thread was currently in the function initializing the data, it will return while the data is still uninitialized. In this case, accessing the first character will cause an access to uninitialized data and result in a StringIndexOutOfBounds exception.

```
for (int i = 0; i < Stonesoup_Str.data.length(); i++) {
    if(Stonesoup_Str.data.charAt(i) >= 'a' || Stonesoup_Str.data.charAt(i) <= 'z'){
        Stonesoup_Str.data.setCharAt(i, (char) (Stonesoup_Str.data.charAt(i) - ('a'
- 'A')));
    }
}

Tracer.tracepointMessage("TRIGGER-POINT: BEFORE");
/* STONESOUP: TRIGGER POINT (double checked lock) */
if(Stonesoup_Str.data.charAt(0) != '\0'){
    output.println(Thread.currentThread().getId() + ": In doStuff2 Stonesoup_Str is:
" + Stonesoup_Str.data.toString());
}
```

If the data is initialized when the call to init_Stonesoup_Str() returns the thread will finish execution without error.

init_Stonesoup_Str()

The init_Stonesoup_Str() function checks to see if the Stonesoup_Str.data is set to null and if not, it locks a mutex lock and checks to see if Stonesoup_Str.data is still null. If the data is still null it will set Stonesoup_Str.data to a new StringBuilder() instance and set the Stonesoup_Str.size to the size of the input string (passed to it by the caller of the function). The function then checks to see if it was passed a control file, and if so reads the control file in order to pause the execution temporarily. This is the crossover point, and allows one thread to create a string object and pause before initializing it. At this point, if another thread were to call the init_Stonesoup_Str() function, it would check to see if Stonesoup_Str.data is null, and since it has already been set to a blank StringBuilder object, return without initializing the data. If the data is then accessed after the thread returns, an uninitialized data access will occur, leading to an exception.

```
if(Stonesoup_Str.data  == null){
    lock.lock();
    if(Stonesoup_Str.data == null){
        Tracer.tracepointMessage("CROSSOVER-POINT: BEFORE");
        /* STONESOUP: CROSS-OVER POINT (double checked lock) */
        Stonesoup_Str.data = new StringBuilder();
        Stonesoup_Str.size = data.length();

        output.println(Thread.currentThread().getId() + ": Initializing second
data");
        if (filename != null) {
            readFile(filename, output);
        }

        Stonesoup_Str.data.append(data);
        Tracer.tracepointMessage("CROSSOVER-POINT: AFTER");
    } else {
        output.println(Thread.currentThread().getId() + ": No need to initialize");
    }
    lock.unlock();
} else {
    output.println(Thread.currentThread().getId() + ": Data is already
initialized");
}
```

Timing Explanation

Due to the random nature of race conditions, we cannot rely on the thread scheduler alone to run our threads and expect that they will deterministically result in either a good or bad outcome. In order to provide a deterministic execution environment we introduced two control mechanisms into the threads themselves. The first is a set of two control files which, when created as FIFO files and controlled by the runFifos.py script described below, ensure that one file read completes two seconds before the other. The other control method is to pass in two regular files for the control files, essentially negating the readFile() commands, and instead passing a significantly large integer in order to influence the thread timing though an array sort on an array of descending integers with size equal to the input integer.

For more information on either of these approaches, view the documentation for multi-threaded race conditions found here.

runFifos.py

The runFifos.py script indirectly controls the time execution order of the threads in a snippet, the specific configuration described below is set by passing two file names of nonexistent files to the runFifos.py script. The script will create both files and control them to ensure that the first FIFO closes two seconds before the second, allowing them to be used to control the execution of the snippet. This is done by writing 32 bytes of data to both FIFOs and waiting for them both to be read, ensuring that both threads in the snippet are synchronized on the file read. The script then writes a second 32 bytes to the first FIFO file and closes it. It then waits two seconds and writes a second 32 bytes to the second FIFO file and closes it as well. The order is as follows:

```
t1: wrote first 32 bytes to file
t2: wrote first 32 bytes to file
ALL: threads synchronized
t1: wrote second 32 bytes to file
t2: t1 finished
t2: wrote second 32 bytes to file
```

The correct execution of this script for the snippets bad test cases is shown below.

```
runFifos.py /opt/stonesoup/workspace/testData/fifo1
/opt/stonesoup/workspace/testData/fifo2
```

Where `/opt/stonesoup/workspace/testData/fifo1` and `/opt/stonesoup/workspace/testData/fifo2` are the names of the control files that will be passed to the snippet when it is executed.

It is worth noting that the runFifos.py script must be executed as a backgrounded pre-process to the snippets execution and allowed to continue to run during the execution of the snippet.

The source code for this script can be found here.

Benign Inputs

Benign inputs come in two forms. The first is using FIFO files with the order passed to the snippet reversed from the order called in the `runFifos.py` script. When executing benign test cases in this format the value of the input integer does not change the execution of the snippet and only serves to slow the overall execution of the snippet when the value is sufficiently large. The second format is to call the snippet with two regular files as control files and a significantly large control integer.

For the following examples that use FIFO files we will assume that the `runFifos.py` script is executed in the format show above.

FIFO Files

Examples of benign inputs with FIFO control files are as follows:

```
50 /opt/stonesoup/workspace/testData/fifo2 /opt/stonesoup/workspace/testData/fifo1
asdf
```

```
50 /opt/stonesoup/workspace/testData/fifo2 /opt/stonesoup/workspace/testData/fifo1
boom boom b0 boom
```

Regular Files

Examples of benign inputs with regular control files are as follows:

```
5000000 /opt/stonesoup/workspace/testData/file1
/opt/stonesoup/workspace/testData/file2 ttrkrkrkttkrk
```

Exploiting Inputs

Exploring inputs all use FIFO control files passed in the same order as they are called in the `runFifos.py` script. Like the benign test case using FIFO control files above, the value of the integer input is irrelevant and simply serves to slow the overall execution of the snippet when the value is sufficiently large. For example:

```
50 /opt/stonesoup/workspace/testData/fifo1 /opt/stonesoup/workspace/testData/fifo2
oops
```

```
50 /opt/stonesoup/workspace/testData/fifo1 /opt/stonesoup/workspace/testData/fifo2
welp, I guess this is goodbye
```

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Restrictions and Requirements

J - CWE-663A - Use of a Non-reentrant Function in a Concurrent Context

Summary

This snippet implements a non-reentrant function that uses a static integer to iterate through a string. The snippet takes a control integer and an input string. The control integer is used for timing within the snippet to ensure that we hit either a good or bad case, and the input string is used as shared data for the threads to act upon. When executing, the snippet spawns two threads which both in turn call a non-reentrant function that iterates through the input string using a static integer as a counter. If both threads enter the function within a significantly small time frame the static integer will get incremented twice for each position in the string leading to an StringIndexOutOfBoundsException.

This is an example of a variant on the Multi-Threaded Race Condition that only uses sorting for timing.

Weakness Class	Concurrency Handling
CWE	CWE-663
Variant	A
Language	Java
Status	**WEAK-363** - CWE-663-01-Java **DONE**
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

This snippet contains two main static classes and a static function: replaceSymbols, toCaps, and arrFunc(), respectively. Both classes are are subclasses of Runnable and run in a thread pool, and the function is used to implement the trigger point. The two classes and the function are described below.

replaceSymbols

The replaceSymbols thread is called with the control integer and the stonesoup output stream. Once executing, the thread allocates an integer array with a size of 500000, and populates the newly created array with integers in descending order (from size to 0). This array is then sorted to ascending order. The purpose of the sort is to slow down the execution of the snippet to allow good test cases to succeed (explained in more detail in the next section). The thread then grabs the mutex lock, and iterates through the shared input string replacing any symbols with '_', and releases the mutex lock.

```
private int threadTiming = 500000;

{...}

int [] sortMe = new int[threadTiming];                    // Timing to prevent the
weakness on good cases
for (int k = 0; k < threadTiming; k++) {
    sortMe[k] = threadTiming - k;
}
Arrays.sort(sortMe);

Tracer.tracepointMessage("replaceSymbols: after qsort");

lock.lock();
char val;
for(int i = 0; i < stonesoup_threadInput.length(); i++){
    val=stonesoup_threadInput.charAt(i);
    if(((val >= '!' && val <= '/') ||
        (val >= ':' && val <= '@') ||
        (val >= '[' && val <= '`') ||
        (val >= '{' && val <= '~')) &&
        (val != '@' && val != '.')) {
            stonesoup_threadInput.setCharAt(i, '_');
    }
}

lock.unlock();
```

Once the thread is finished replacing symbols it calls arrFunc() with the control integer, the name of a file, and the output stream. This is the crossover point.

```
/* STONESOUP: CROSS-OVER POINT (non-reentrant function) */
arrFunc(size, "/opt/stonesoup/workspace/testData/replace.txt", output);
```

After arrFunc() returns, assuming that the weakness was not hit, the thread completes execution and joins the main thread without error.

toCaps

The toCaps thread is called with the control integer and the stonesoup output stream. Once executing, the thread grabs the mutex lock, iterates through the shared input string converting all characters to capitals, and releases the mutex lock.

```
lock.lock();

for (int i = 0; i < stonesoup_threadInput.length(); i++) {
    if(stonesoup_threadInput.charAt(i) >= 'a' || stonesoup_threadInput.charAt(i) <=
'z'){
        stonesoup_threadInput.setCharAt(i, (char) (stonesoup_threadInput.charAt(i) -
('a' - 'A')));
    }
}
lock.unlock();
```

Once the thread has finished converting the string to capitals, it calls arrFunc() with the control integer, a file name, and the output stream.

This is also a crossover point.

```
/* STONESOUP2: CROSS-OVER POINT (non-reentrant function) */
arrFunc(size, "/opt/stonesoup/workspace/testData/toCaps.txt", output);
```

After arrFunc() returns, assuming that the weakness was not hit, the thread completes execution and joins the main thread without error.

arrFunc()

arrFunc() is a static function the contains the trigger point for the weakness. arrFunc() takes in a control integer, a file name, and an output stream. The control integer is used for timing, and the larger it is, the more likely the weakness will be hit. The file name is the name of a file to log data to encourage context switching in the bad case, and the output stream is used simply to log data.

Once the function has been called, it allocates an integer array with size equal to the control integer, and populates the newly created array with integers in descending order (from size to 0). This array is then sorted to ascending order in order to slow down the execution of the function. The function then enters a for loop that iterates through the shared input string, incrementing a global static integer 'j' that is used to index the string. This indexing of the string is the trigger point. If two threads are in this function simultaneously 'j' will get incremented twice per iteration of the for loop causing it to over-read the string resulting in a StringIndexOutOfBoundsException.

```
int[] sortMe = new int[size];
j = 0;
Tracer.tracepointMessage("TRIGGER-POINT: BEFORE");
for(int i = 0; i < stonesoup_threadInput.length(); i++, j++){
    /* STONESOUP: TRIGGER POINT (non-reentrant function) */
    stonesoup_threadInput.setCharAt(j, '\0');                    // Increment by
static variable, if two threads are in the
                                                                //    function
incrementing j, the variable will retain both
                                                                //    increments,
casing an StringIndexOutOfBoundsException
    output.format("TID: %d I: %d J: %d\n", Thread.currentThread().getId(), i, j);

    if (size > 5) {
        try {

            PrintWriter fileoutput = new PrintWriter(new BufferedWriter(new
FileWriter(tempfile)));
            fileoutput.println("Iteration: " + i);
            fileoutput.close();
        } catch (IOException e) {
            Tracer.tracepointError("IOException");
            // Do nothing
        }
        for (int k = 0; k < size; k++){                               // Timing to
ensure the weakness occurs
            sortMe[k]=size-k;
        }
        Arrays.sort(sortMe);
    }
}
```

Each iteration of the loop is slowed down by an additional array sort in order to provide a larger gap for the two threads to operate in when executing a bad test case.

If two threads are not executing simultaneously, the function will perform as expected and an exception will not get thrown.

Timing Explanation

Due to the random nature of race conditions, we cannot rely on the thread scheduler alone to run our threads and expect that they will deterministically result in either a good or bad outcome. In order to provide a deterministic execution environment we introduce a control mechanism into the threads themselves. This control mechanism is a series if array sorts that are used to slow down certain threads in order to allow the other thread(s) time to execute while the other is in a certain state.

For more information on timing approaches, view the documentation for multi-threaded race conditions found here.

Benign Inputs

Benign inputs all use a significantly small control integer to allow the threadTiming in replaceSymbols to slow down that thread for long enough that toCaps can return from arrFunc() before replaceSymbols enters it. For example:

```
2 a ab abc cba ba a
```

```
5 jajajajajajajajajaja
```

```
3 55 in the hive
```

Exploiting Inputs

Benign inputs all use a significantly large control integer in order to allow time for both threads to enter arrFunc() at relatively the same time, causing the weakness to occur. For example:

```
5000000 8675309
```

```
8000000 skippy lue
```

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Restrictions and Requirements

J - CWE-764A - Multiple Locks of a Critical Resource

Summary

This snippet implements two threads that lock a reentrant lock multiple times, causing a deadlock if the lock is locked more times that it is unlocked. The snippet takes a string as input and spawns two threads that both iterate through the string and lock a reentrant lock for each '1' encountered in the string. If there are three or more '1's in the string, the two unlocks at the end of the thread's execution will leave the reentrant lock in a locked state, causing the second thread to hang indefinitely on the first lock.

Weakness Class	Concurrency Handling

CWE	CWE-764
Variant	A
Language	Java
Status	WEAK-365 - CWE-764 -01-Java DONE
Negative Technical Impact(s)	DOS: Deadlock

Implementation

This snippet contains one static class, HelloRunnable, which is a subclass of runnable, and is run by two threads in a thread pool simultaneously. The class is passed the input string and once executing, starts iterating through the string and upon encountering the first '1', grabs a reentrant lock and breaks out of the loop saving it's current index. Once the first thread has locked the reentrant lock, the second thread will hang trying to grab the lock until the first has properly released it.

```
int index = 0;
while (index < input.length()) {
    char cc = input.charAt(index);
    index++;
    if (cc == '1') {
        lock.lock();
        break;
    }
}
```

The thread then iterates through the rest of the string counting the number of characters between the first and second '1', locking the reentrant lock each time it encounters another '1'. If the thread encounters more than one more '1', the lock is locked three or more times, which will cause the second thread to deadlock at the first lock, shown above. This is both the crossover and trigger point.

```
boolean found1 = false;
while (index < input.length()) {
    char cc = input.charAt(index);
    index++;
    if (!found1) {
        count++;
    }
    if (cc == '1') {
        /* STONESOUP: CROSS-OVER POINT (Multiple locks of a critical resource) */
        /* STONESOUP: TRIGGER POINT (Multiple locks of a critical resource) */
        lock.lock();
        found1 = true;
    }
}
```

Once the string iteration has finished, the snippet unlocks the reentrant lock, up to two times if it has been locked multiple times, and exits, printing the number of characters between the first and second '1'.

```
if (lock.isHeldByCurrentThread()) {
    if (lock.getHoldCount() > 1) {
        lock.unlock();
    }
    lock.unlock();
}
output.println(
        "Info: Found " + getCount() + " letters between 1 and 1");
```

If the lock was only locked once or twice, the second thread will then proceed to follow the same execution path as the first. Otherwise, if the lock was locked three or more times, the second thread will deadlock on the first attempt to lock the reentrant lock.

Benign Inputs

Benign inputs all contain less than three '1's. For example:

```
1234
```

```
21212
```

```
0010001
```

Exploiting Inputs

Exploiting inputs all contain three or more '1's. For example:

```
111
```

```
123412341
```

Source Code

For more information, please refer to the weakness Source Code.

Related Information

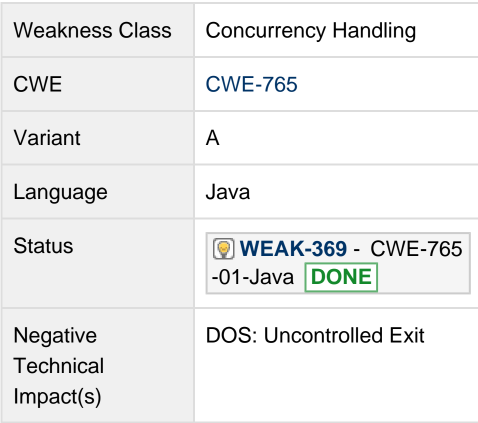
- Execution Environment
- Restrictions and Requirements

J - CWE-765A - Multiple Unlocks of a Critical Resource

Summary

This snippet implements a thread that locks a reentrant lock, and in certain circumstances unlocks the reentrant lock more than once causing an exception. The snippet takes a string as input and spawns a threads that iterates through the string and locks a reentrant lock when it encounters the first '1' in the string. For every '1' after that the thread unlocks the reentrant lock. If there are more than two '1's in the input the thread unlocks the reentrant lock multiple times causing an exception.

This snippet differs from the snippet for CWE-832 because this snippet unlocks a previously locked resource multiple times and the snippet for CWE-832 unlocks a resource that has not been previously locked.



Weakness Class	Concurrency Handling
CWE	CWE-765
Variant	A
Language	Java
Status	**WEAK-369** - CWE-765 -01-Java DONE
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

This snippet contains one static class, HelloRunnable, which is a subclass of runnable, and is run in a thread pool. The class is passed the input string and once executing, starts iterating through the string and upon encountering the first '1', grabs a reentrant lock and breaks out of the loop saving it's current index.

```
int index = 0;
while (index < input.length()) {
    char cc = input.charAt(index);
    index ++;
    if (cc == '1') {
        Tracer.tracepointMessage("Locking lock");
        Tracer.tracepointVariableInt("index", index);
        lock.lock();
        break;
    }
}
```

The thread then iterates through the rest of the string counting the number of characters between the first and second '1', unlocking the reentrant lock each time it encounters another '1'. If the thread encounters more than one more '1', the lock is unlocked multiple times causing an exception. This is both the crossover and the trigger point.

```
boolean found1 = false;
while (index < input.length()) {
    char cc = input.charAt(index);
    index ++;
    if (!found1) {
        count++;
    }
    if (cc == '1') {
        /* STONESOUP: CROSS-OVER POINT (Multiple unlocks of a critical resource) */
        /* STONESOUP: TRIGGER POINT (Multiple unlocks of a critical resource) */
        Tracer.tracepointMessage("Unlocking lock");
        lock.unlock();
        found1 = true;
    }
}
```

Once the string iteration has finished, the snippet unlocks the reentrant lock if it is still currently held and finishes execution.

```
if (lock.isHeldByCurrentThread()) {
    Tracer.tracepointMessage("Unlocking lock");
    lock.unlock();
}
```

Benign Inputs

Benign inputs all contain less than three '1's. For example:

```
1234
```

```
21212
```

```
0010001
```

Exploiting Inputs

Exploiting inputs all contain three or more '1's. For example:

```
111
```

```
123412341
```

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Restrictions and Requirements

J - CWE-820A - Missing Synchronization

Summary

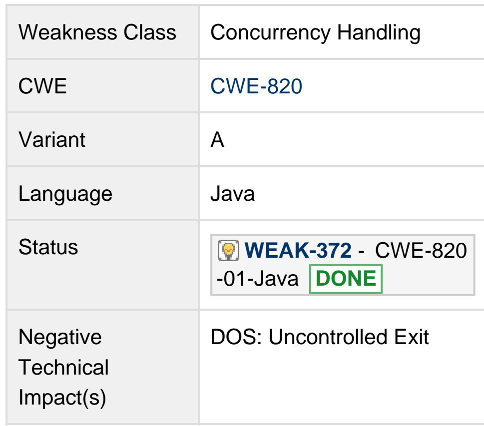
This snippet implements two threads that do not use synchronization while accessing a shared resource. The snippet takes a control integer, the names of two control files, and an input string. The control integer and the two control files are used for timing within the snippet to ensure that we hit either a good or bad case, and the input string is used as shared data for the threads to act upon. When executing, the snippet spawns two threads, one of which calculates an increment amount, and the other of which uses this increment amount to iterate through the shared input string. If the calculated increment amount is negative, and the thread that uses it to iterate though the string does so before it is sanitized, the snippet will throw a StringIndexOutOfBounds exception.

This is an example of a Multi-Threaded Race Condition.

This snippet differs from the snippet for CWE-414 because this snippet spawns two threads and neither thread implements a locking mechanism while both access a shared object and the snippet for CWE-414 implements two threads that access a shared object but only one of them implements a locking mechanism.

This snippet differs from the snippet for CWE-567 because the snippet for CWE-567 spawns two thread and neither thread implements a locking mechanism while both access a static shared variable. It should be noted that this snippet is very similar to the snippet for CWE-567 because CWE-820 is a parent of CWE-567. The primary difference between the two snippets resides in the use of a shared object in the case of this snippet and a shared static variable in the case of the snippet for CWE-820.

This snippet differs from the snippet for CWE-821 because the snippet for CWE-821 spawns two threads which each use a separate lock object.



Weakness Class	Concurrency Handling
CWE	CWE-820
Variant	A
Language	Java
Status	WEAK-372 - CWE-820 -01-Java DONE
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

This snippet contains two main static classes: CalculateIncrementAmount, and ConvertToPound, both of which are are subclasses of Runnable and run in a thread pool. The two classes are described below.

CalculateIncrementAmount

The CalculateIncrementAmount thread is called with the input string, the second control file, and the stonesoup output stream. Once executing, the thread calculates a global increment based on the first character of the input string, and reads the control file (used for thread

timing, described in the next section). This is the crossover point.

```
/* STONESOUP: CROSS-OVER POINT (missing sync) */
threadInput.increment = threadInput.data.charAt(0) - 'A';
Tracer.tracepointVariableInt("threadInput.increment", threadInput.increment);
Tracer.tracepointMessage("CROSSOVER-POINT: AFTER");

readFile(filename, output);
```

Once the control file is finished being read, assuming the weakness has not occurred, the thread sanitizes the global increment.

```
if (this.threadInput.increment < 0) {
    this.threadInput.increment *= -1;
} else if (this.threadInput.increment == 0) {
    this.threadInput.increment += 1;
}
```

The thread then completes its execution and joins the main thread without error.

ConvertToPound

The ConvertToPound thread is called with the input string, the control integer, the first control file, and the stonesoup output stream. Once executing, the thread allocates an integer array with size equal to the control integer, and populates the newly created array with integers in descending order (from size to 0). This array is then sorted to ascending order. The purpose of the sort is to slow down the execution of the snippet in certain good test cases (explained in more detail in the next section). Once the array sort is finished, the thread reads it's control file (again to control the execution time of the thread, explained below), and iterates through the shared input string, using the global increment as the increment for it's loop counter. If the increment is currently set to a negative number this loop will result in a StringIndexOutOfBounds exception being thrown. This is the trigger point.

```
for (int i = 0; i < this.size; i++) {
    sortMe[i] = this.size - i;
}
Arrays.sort(sortMe);

readFile(filename, output);
Tracer.tracepointMessage("TRIGGER-POINT: BEFORE");
Tracer.tracepointVariableInt("threadInput.increment", threadInput.increment);
/* STONESOUP: TRIGGER POINT (missing sync) */
for (int i = 0; i < this.threadInput.data.length(); i += this.threadInput.increment)
{
    this.threadInput.data.setCharAt(i, '#');
}
```

If the increment is positive, the loop will terminate properly and the thread will complete execution, joining the main thread without error.

Timing Explanation

Due to the random nature of race conditions, we cannot rely on the thread scheduler alone to run our threads and expect that they will deterministically result in either a good or bad outcome. In order to provide a deterministic execution environment we introduced two control mechanisms into the threads themselves. The first is a set of two control files which, when created as FIFO files and controlled by the runFifos.py script described below, ensure that one file read completes two seconds before the other. The other control method is to pass in two regular files for the control files, essentially negating the readFile() commands, and instead passing a significantly large integer in order to influence the thread timing though an array sort on an array of descending integers with size equal to the input integer.

For more information on either of these approaches, view the documentation for multi-threaded race conditions found here.

runFifos.py

The runFifos.py script indirectly controls the time execution order of the threads in a snippet, the specific configuration described below is set by passing two file names of nonexistent files to the runFifos.py script. The script will create both files and control them to ensure that the first FIFO closes two seconds before the second, allowing them to be used to control the execution of the snippet. This is done by writing 32 bytes of data to both FIFOs and waiting for them both to be read, ensuring that both threads in the snippet are synchronized on the file read. The script then writes a second 32 bytes to the first FIFO file and closes it. It then waits two seconds and writes a second 32 bytes to the second FIFO file and closes it as well. The order is as follows:

```
t1: wrote first 32 bytes to file
t2: wrote first 32 bytes to file
ALL: threads synchronized
t1: wrote second 32 bytes to file
t2: t1 finished
t2: wrote second 32 bytes to file
```

The correct execution of this script for the snippets bad test cases is shown below.

```
runFifos.py /opt/stonesoup/workspace/testData/fifo1
/opt/stonesoup/workspace/testData/fifo2
```

Where `/opt/stonesoup/workspace/testData/fifo1` and `/opt/stonesoup/workspace/testData/fifo2` are the names of the control files that will be passed to the snippet when it is executed.

It is worth noting that the runFifos.py script must be executed as a backgrounded pre-process to the snippets execution and allowed to continue to run during the execution of the snippet.

The source code for this script can be found here.

Benign Inputs

Benign inputs come in two forms. The first is using FIFO files with the order passed to the snippet reversed from the order called in the `runFifos.py` script. When executing benign test cases in this format the value of the input integer does not change the execution of the snippet and only serves to slow the overall execution of the snippet when the value is sufficiently large. The second format is to call the snippet with two regular files as control files and a significantly large control integer.

For the following examples that use FIFO files we will assume that the `runFifos.py` script is executed in the format show above.

FIFO Files

Examples of benign inputs with FIFO control files are as follows:

```
50 /opt/stonesoup/workspace/testData/fifo2 /opt/stonesoup/workspace/testData/fifo1
1abcdabcd
```

```
50 /opt/stonesoup/workspace/testData/fifo2 /opt/stonesoup/workspace/testData/fifo1
somevalue
```

Regular Files

Examples of benign inputs with regular control files are as follows:

```
5000000 /opt/stonesoup/workspace/testData/file1
/opt/stonesoup/workspace/testData/file2 stringvalue
```

Exploiting Inputs

Exploring inputs all use FIFO control files passed in the same order as they are called in the `runFifos.py` script. Like the benign test case using FIFO control files above, the value of the integer input is irrelevant and simply serves to slow the overall execution of the snippet when the value is sufficiently large. For example:

```
50 /opt/stonesoup/workspace/testData/fifo1 /opt/stonesoup/workspace/testData/fifo2
123
```

```
50 /opt/stonesoup/workspace/testData/fifo1 /opt/stonesoup/workspace/testData/fifo2
1stnumb3r
```

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Restrictions and Requirements

J - CWE-821A - Incorrect Synchronization

Summary

This snippet implements two threads that each use a separate reentrant lock object while accessing a shared resource. The snippet takes a control integer, the names of two control files, and an input string. The control integer and the two control files are used for timing within the snippet to ensure that we hit either a good or bad case, and the input string is used as shared data for the threads to act upon. When executing, the snippet spawns two threads, one of which calculates a divisor based on the input string, and the other of which uses that divisor to divide each character in the string. If the divisor is calculated to be 0 and the second thread uses this value before it is sanitized the snippet will throw a DivideByZero exception.

This is an example of a Multi-Threaded Race Condition.

This snippet differs from the snippet for CWE-414 because this snippet spawns two threads which each use a separate lock object and the snippet for CWE-414 implements two threads that access a shared object but only one of them implements a locking mechanism.

This snippet differs from the snippet for CWE-567 because the snippet for CWE-567 spawns two thread and neither thread implements a locking mechanism while both access a static shared variable.

This snippet differs from the snippet for CWE-820 because the snippet for CWE-820 spawns two threads and neither thread implements a locking mechanism while both access a shared object.

Weakness Class	Concurrency Handling
CWE	CWE-821
Variant	A
Language	Java
Status	WEAK-374 - CWE-821-01-Java DONE
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

This snippet contains two main static classes: calcDevAmount, and devChar, both of which are are subclasses of Runnable and run in a thread pool. The two classes are described below.

calcDevAmount

The calcDevAmount thread is called with a shared object for storing the divisor, the second control file, and the stonesoup output stream. It also has access to the shared input string in a static variable 'stonesoup_threadInput'. Once executing, the thread locks a reentrant lock called 'lock' and calculates a divisor from the first character of the input string. This is the crossover point.

```
lock.lock();

Tracer.tracepointMessage("CROSSOVER-POINT: BEFORE");
/* STONESOUP: CROSS-OVER POINT (Incorrect Synchronization) */
dev_amount.setVal(stonesoup_threadInput.charAt(0) - 'A');
```

The thread then reads the control file (used to control the thread execution time, explained in the next section), and once the control file read is finished, assuming the weakness has not been executed, the thread sanitizes the divisor.

```
readFile(filename, output);

if (dev_amount.getVal() < 0) {
    dev_amount.setVal(dev_amount.getVal() * -1);
}
if (dev_amount.getVal() == 0) {
    dev_amount.setVal(dev_amount.getVal() + 1);
}
```

The thread then completes its execution, joining the main thread without error.

devChar

The devChar thread is called with a shared object for storing the divisor, the control integer, the first control file, and the stonesoup output stream. It also has access to the shared input string in a static variable 'stonesoup_threadInput'. Once executing, the thread locks a second reentrant lock called 'lock2' and allocates an integer array with size equal to the control integer, populating the newly created array with integers in descending order (from size to 0). This array is then sorted to ascending order. The purpose of the sort is to slow down the execution of the snippet in certain good test cases (explained in more detail in the next section). Once the array sort is finished, the thread reads it's control file (again to control the execution time of the thread, explained below), and iterates through the shared string dividing each character with the divisor calculated by the other thread. If the divisor is zero, this will result in a DivideByZero exception. This is the trigger point.

```
lock2.lock();
int[] sortMe = new int[size];
for (int i = 0; i < size; i++) {
    sortMe[i] = size - i;
}
Arrays.sort(sortMe);

readFile(filename, output);

Tracer.tracepointMessage("TRIGGER-POINT: BEFORE");
Tracer.tracepointVariableInt("dev_amount.getVal()", dev_amount.getVal());
for (int i = 0; i < stonesoup_threadInput.length(); i++) {
    /* STONESOUP: TRIGGER POINT (Incorrect Synchronization) */
    stonesoup_threadInput.setCharAt(i, (char)(stonesoup_threadInput.charAt(i) /
dev_amount.getVal()));
}
Tracer.tracepointMessage("TRIGGER-POINT: AFTER");
lock2.unlock();
```

If the divisor is a positive non-zero number, the thread will finish executing and join the main thread without error.

Timing Explanation

Due to the random nature of race conditions, we cannot rely on the thread scheduler alone to run our threads and expect that they will deterministically result in either a good or bad outcome. In order to provide a deterministic execution environment we introduced two control mechanisms into the threads themselves. The first is a set of two control files which, when created as FIFO files and controlled by the runFifos.py script described below, ensure that one file read completes two seconds before the other. The other control method is to pass in two regular files for the control files, essentially negating the readFile() commands, and instead passing a significantly large integer in order to influence the thread timing though an array sort on an array of descending integers with size equal to the input integer.

For more information on either of these approaches, view the documentation for multi-threaded race conditions found here.

runFifos.py

The runFifos.py script indirectly controls the time execution order of the threads in a snippet, the specific configuration described below is set by passing two file names of nonexistent files to the runFifos.py script. The script will create both files and control them to ensure that the first FIFO closes two seconds before the second, allowing them to be used to control the execution of the snippet. This is done by writing 32 bytes of data to both FIFOs and waiting for them both to be read, ensuring that both threads in the snippet are synchronized on the file read. The script then writes a second 32 bytes to the first FIFO file and closes it. It then waits two seconds and writes a second 32 bytes to the second FIFO file and closes it as well. The order is as follows:

```
t1: wrote first 32 bytes to file
t2: wrote first 32 bytes to file
ALL: threads synchronized
t1: wrote second 32 bytes to file
t2: t1 finished
t2: wrote second 32 bytes to file
```

The correct execution of this script for the snippets bad test cases is shown below.

```
runFifos.py /opt/stonesoup/workspace/testData/fifo1
/opt/stonesoup/workspace/testData/fifo2
```

Where `/opt/stonesoup/workspace/testData/fifo1` and `/opt/stonesoup/workspace/testData/fifo2` are the names of the control files that will be passed to the snippet when it is executed.

The runFifos.py script must be executed as a backgrounded pre-process to the snippet's execution and allowed to continue to run during the execution of the snippet.

The source code for this script can be found here.

Benign Inputs

Benign inputs come in two forms. The first is using FIFO files with the order passed to the snippet reversed from the order called in the `runFifos.py` script. When executing benign test cases in this format the value of the input integer does not change the execution of the snippet and only serves to slow the overall execution of the snippet when the value is sufficiently large. The second format is to call the snippet with two regular files as control files and a significantly large control integer.

For the following examples that use FIFO files we will assume that the `runFifos.py` script is executed in the format show above.

FIFO Files

Examples of benign inputs with FIFO control files are as follows:

```
50 /opt/stonesoup/workspace/testData/fifo2 /opt/stonesoup/workspace/testData/fifo1
abcdabcd
```

```
50 /opt/stonesoup/workspace/testData/fifo2 /opt/stonesoup/workspace/testData/fifo1
somevalue
```

Regular Files

Examples of benign inputs with regular control files are as follows:

```
5000000 /opt/stonesoup/workspace/testData/file1
/opt/stonesoup/workspace/testData/file2 Astringvalue

Where /opt/stonesoup/workspace/testData/file1 and
/opt/stonesoup/workspace/testData/file2 are pre-existing empty files.
```

Exploiting Inputs

Exploring inputs all use FIFO control files passed in the same order as they are called in the `runFifos.py` script, and must have the character of the input string as an 'A'. Like the benign test case using FIFO control files above, the value of the integer input is irrelevant and simply serves to slow the overall execution of the snippet when the value is sufficiently large. For example:

```
50 /opt/stonesoup/workspace/testData/fifo1 /opt/stonesoup/workspace/testData/fifo2
ABC123
```

```
50 /opt/stonesoup/workspace/testData/fifo1 /opt/stonesoup/workspace/testData/fifo2
Anumb3rYO
```

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Restrictions and Requirements

J - CWE-832A - Unlock of a Resource that is not Locked

Summary

The snippet implements a a thread which locks one of two reentrant locks, and then unlocks the second one. If the first lock was locked, unlocking the second will cause an unlock of a resource that is not locked, leading to an exception. This snippet takes a single string as input, and uses this string to determine which lock it locks. If the string contains an 'A' before any 'a's the first lock will be locked and an exception will occur, however, if the string contains an 'a' before any 'A's the second lock will be locked and the snippet will execute properly.

This snippet differs from the snippet for CWE-765 because this snippet unlocks a resource that has not been previously locked and the snippet for CWE-765 unlocks a previously locked resource multiple times.

Weakness Class	Concurrency Handling
CWE	CWE-832
Variant	A
Language	Java
Status	**WEAK-384** - CWE-832 -01-Java DONE
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

This snippet consists of one static class, HelloRunnable, that is a subclass of Runnable and executed in a thread pool. This thread is called with the input string and the stonesoup output stream. When executing, the thread iterates through the input string and upon finding the first 'a' or 'A', calls the lockA() method.

```
while (index < input.length()) {
char cc = input.charAt(index);
index ++;
if (Character.toUpperCase(cc) == 'A') {
    lockA(cc);
    break;
}
```

The lockA() method locks upperLock if the character is an 'A' and locks lowerLock if the character is an 'a'. This is the crossover point.

```
private void lockA(Character cc) {
    Tracer.tracepointLocation("STONESOUP_INJECTED_FILE", "HelloRunable.lockA");
    Tracer.tracepointMessage("CROSSOVER-POINT: BEFORE");
    /* STONESOUP: CROSS-OVER POINT (Unlock of a Resource that is not Locked) */
    if (Character.isUpperCase(cc)) {
        Tracer.tracepointMessage("Locking upperLock");
        upperLock.lock();
    } else {
        Tracer.tracepointMessage("Locking lowerLock");
        lowerLock.lock();
    }
    Tracer.tracepointMessage("CROSSOVER-POINT: AFTER");
}
```

The thread then continues to iterate though the string from where it left off and upon encountering the next 'a' or 'A' call's unlockA().

```
while (index < input.length()) {
    char cc = input.charAt(index);
    index++;
    if (Character.toUpperCase(cc) == 'A') {
        unlockA(cc);
        break;
    } else {
        count++;
    }
}
```

unlockA() unlocks lowerLock, leading to an unlock of a resource that isn't locked if upperLock was previously locked instead of lowerLock. This is the trigger point.

```
private void unlockA(Character cc) {
    Tracer.tracepointLocation("STONESOUP_INJECTED_FILE", "HelloRunable.unlockA");
    Tracer.tracepointMessage("TRIGGER-POINT: BEFORE");
    /* STONESOUP: TRIGGER POINT (Unlock of a Resource that is not Locked) */
    Tracer.tracepointMessage("Unlocking lowerlock");
    lowerLock.unlock();
    Tracer.tracepointMessage("TRIGGER-POINT: AFTER");
}
```

If lowerLock had been previously locked, the thread finishes execution and joins the main thread without error.

Benign Inputs

Benign inputs all contain an 'a' before an 'A' character. For example:

```
abcdabcd
```

```
abababa
```

```
abABababab
```

Exploiting Inputs

Exploiting inputs all contain an 'A' before an 'a' character. For example:

```
ABab
```

```
04A04a
```

Source Code

For more information, please refer to the weakness Source Code.

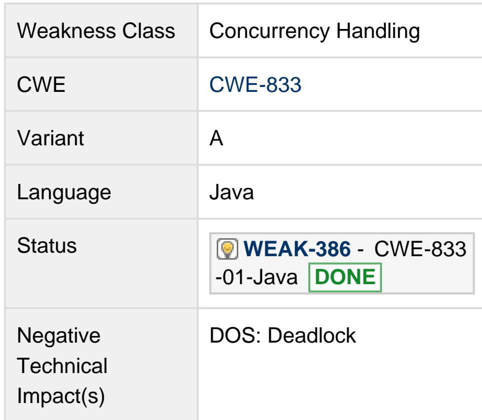
Related Information

- Execution Environment
- Restrictions and Requirements

J - CWE-833A - Deadlock

Summary

This snippet implements two reentrant locks called upperLock and lowerLock such that, if the main process locks the same lock that a thread it spawns uses the snippet will result in deadlock. The snippet takes a string as input, and if the first character of the string is a lower case letter, the main thread will lock lowerLock and spawn a thread that uses upperLock. However, if the first character is an upper case letter, the main thread will lock upperLock and spawn a thread that uses upperLock resulting in deadlock.



Weakness Class	Concurrency Handling
CWE	CWE-833
Variant	A
Language	Java
Status	WEAK-386 - CWE-833-01-Java DONE
Negative Technical Impact(s)	DOS: Deadlock

Implementation

The snippet consists of the code executed in the weakness() function and a static class called CountUpper that is a subclass of Runnable and executed in a thread pool. The snippet implements two reentrant locks called upperLock and lowerLock, and starts by taking in a string as input. The snippet then locks either upperLock or lowerLock based on whether the first character of the input string is an upper case or lower case letter respectively. This is the crossover point.

```
/* STONESOUP: CROSS-OVER POINT (Deadlock) */
if (stonesoup_upper) {
    Tracer.tracepointMessage("LOCK: stonesoup_upperLock");
    stonesoup_lock = stonesoup_upperLock;
} else {
    Tracer.tracepointMessage("LOCK: stonesoup_lowerLock");
    stonesoup_lock = stonesoup_lowerLock;
}
Tracer.tracepointMessage("Locking lock");
stonesoup_lock.lock();
```

The snippet then proceeds to spawn a thread that attempts to grab upperLock. If the main thread already holds upperLock this will cause the thread to hang indefinitely, and since the thread will not return, the main thread will hang indefinitely was well, waiting for the thread to join. This is the trigger point.

```
// Inside CountUpper
/* STONESOUP: TRIGGER POINT (Deadlock) */
Tracer.tracepointMessage("Locking lock");
stonesoup_upperLock.lock();
```

```
// Inside weakness() main thread hangs here
stonesoup_thread1.join();
```

If the main thread grabbed lowerLock instead, the snippet will execute properly and return without error.

Benign Inputs

Benign inputs all start with a lowercase character. For example:

```
add
```

```
a04f
```

```
aa
```

Exploiting Inputs

Exploiting inputs all start with an uppercase character. For example:

```
A04F
```

```
Add
```

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Restrictions and Requirements

Error Handling

This class contains weaknesses related to the handling of errorsl. These weaknesses involve:

- issues that arise from improper handling of errors

Weakness Members

The error handling weakness class is defined as a set of weakness types as defined by the MITRE Common Weakness Enumeration (CWE) ontology. The MITRE CWE ontology is defined by a tree, and as such may target CWEs that are a parent or child of another CWE. In those cases, CWEs may be proposed to remove from the test set target given that they are fully covered by one or more other CWEs. Individual weakness variants (i.e. tests) are designed around the definition, description, and code examples provided for each CWE.

CWE	Name	Target Source Languages
CWE-209	Information Exposure Through and Error Message	Java
CWE-248	Uncaught Exception	Java
CWE-252	Unchecked Return	Java
CWE-253	Incorrect Check of Function Return Value	Java
CWE-390	Detection of Error Condition Without Action	Java
CWE-391	Unchecked Error Condition	Java
CWE-460	Improper Cleanup on Thrown Exception	Java
CWE-584	Return Inside Finally	Java

Weakness Variants

The following weakness variants are developed and for the C and Java source languages.

Java Source Language

Error rendering macro 'detailssummary' : In template Confluence.Templates.Macro.MasterDetail.contentLink2: When evaluating "contextPath()": Error while computing function "contextPath()": null

Runtime Information

Notes

J - CWE-209A - Information Exposure Through An Error Message

Summary

This snippet attempts to connect to a MYSQL database and insert a record. If an error occurs while inserting the record. Sensitive information about the database and the connection is printed to the screen. This can be triggered by providing input that creates a record that is too long to fit within the tables size parameters.

Weakness Class	Error Handling
CWE	CWE-209
Variant	A
Language	Java
Status	WEAK-186 - CWE-209 -01-Java DONE
Negative Technical Impact(s)	Read Application Data

Implementation

The snippet sets up a standard database connection to a MYSQL database. It then takes a companyName as input and checks to see if it is null.

```
if (companyName == null) {
     output.println("No company name provided.");
}
```

If the companyName isn't null, then the snippet attempts to use it in an insert query to the database.

```
else {
    try {
         PreparedStatement stmt = con.prepareStatement("INSERT INTO Shippers
(CompanyName, Phone) VALUES (?, ?)");
         /* STONESOUP: crossover point (user controlled value in query) */
         stmt.setString(1, companyName);
         stmt.setNull(2, Types.NULL);
         /* STONESOUP: trigger point */
         if (stmt.executeUpdate() > 0) {
              output.println("Shipper added successfully.");
         } else {
              output.println("No rows added.");
         }
    }
```

If a SQL exception is caught, then sensitive information is printed out.

```
catch (SQLException se) {
     output.println("Database Error!");
     output.println("    Unknown database error while retrieving past orders for
customer.");
     output.println("");
     output.println("Connection Details");
     output.printf("    Host: %s\n", stonesoup_mysql_host);
     output.printf("    Port: %s\n", stonesoup_mysql_port);
     output.printf("    User: %s\n", stonesoup_mysql_user);
     output.printf("    Pass: %s\n", stonesoup_mysql_pass);
     output.printf("    JDBC: %s\n", stonesoup_jdbc);
     output.println("");
     output.println("Error Message");
     output.println(se.getMessage());
     output.println("");
     output.println("Stacktrace");
     se.printStackTrace(output);
}
```

Benign Inputs

Any company name that fits the size requirements for the database. For example:

```
IBM
```

```
Digital
```

Due to the fact that none of these values are large enough to cause an integer overflow, they will not result in a bad state.

Exploiting Inputs

Any company name that is too large to fit into the database table. For example:

```
DigitalPhotographicalTecnologicalSprectrumAnalysis
```

```
AutomatedTellerMachineServicingPartsAndRepairCenter
```

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Restrictions and Requirements

J - CWE-248A - Uncaught Exception

Summary

This snippet takes an input and attempts to convert it to upper case. If the value contains anything other than a-zA-Z then an exception is thrown, but not caught. Thus resulting in an uncaught exception.

Weakness Class	Error Handling
CWE	CWE-248
Variant	A
Language	Java
Status	WEAK-188 - CWE-248 -01-Java DONE
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

The snippet converts the input string to a character array.

```
char[] stonesoup_characters = value.toCharArray();
```

The snippet then tries to loop over the array and convert it to uppercase.

```
for(int i=0; i < stonesoup_characters.length; i++){
     if(Character.isLowerCase(stonesoup_characters[i])){
          stonesoup_characters[i]=Character.toUpperCase(stonesoup_characters[i]);
     }
```

If a character is found that isn't able to be converted to uppercase, or isn't uppercase, then an exception is thrown but not caught.

```
else if(!Character.isUpperCase(stonesoup_characters[i])){
     /* STONESOUP: CROSSOVER POINT */
     throw new java.lang.IllegalArgumentException("Not a valid character");
}
}
```

This exception is caught by a default exception handler, so that we can observe the results.

```
Thread.setDefaultUncaughtExceptionHandler(new Thread.UncaughtExceptionHandler() {
     // @Override - removed due to Java 1.4
     public void uncaughtException(Thread t, Throwable e) {
          /* STONESOUP: TRIGGER POINT */
          System.err.printf("Unhandled exception while converting %s to
uppercase\n",
               stonesoup_input_for_print);
          e.printStackTrace(System.err);
     }
});
```

Benign Inputs

Any string that contains only characters a-zA-Z. For example:

```
SomeValue
```

```
ALREADYUPPERCASE
```

```
alllowercase
```

Exploiting Inputs

Any string that contains characters outside the range of a-zA-Z. For example:

```
c0ntainsnumb3rs
```

```
"containspunctuation!"
```

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Restrictions and Requirements

J - CWE-252A - Unchecked Return

Summary

This snippet attempts to convert a string to upper case. If the value contains characters not within the range of a-zA-Z then the function used to convert the string to upper case will return null. The functions return value is then compared to a static 'password'. If the uppercase function returned null, then the comparison will cause an error.

Weakness Class	Error Handling
CWE	CWE-252
Variant	A
Language	Java
Status	**WEAK-191** - CWE-252 -01-Java **DONE**
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

The snippet attempts to convert the input to uppercase. If the input contains characters that aren't within the range of a-zA-Z then the function will return null.

```
public static String stonesoup_to_upper(final String input) {
     char stonesoup_char = 0;
     String retval = input;
     for (int i = 0; i < retval.length(); i++) {
          stonesoup_char = retval.charAt(i);
          if (Character.isLowerCase(stonesoup_char)) {
               retval = retval.replace(stonesoup_char,
               Character.toUpperCase(stonesoup_char));
          } else if (!Character.isUpperCase(stonesoup_char)) {
               return null;
          }
     }
     return retval;
}

/* STONESOUP: CROSSOVER POINT */
String capitalized_value = stonesoup_to_upper(value);
```

The snippet then compares the returned value to a static password. If the returned value is null, this will result in an exception.

```
String password = "STONESOUP";
try {
     /* STONESOUP: TRIGGER POINT */
     if(    password.compareTo(capitalized_value)==0){
         output.println("passwords match");
     }
} catch (NullPointerException e) {
     e.printStackTrace(output);
     throw e;
}
```

Benign Inputs

Any value that only contains a-zA-Z. For example:

```
alllowercase
```

```
stonesoup
```

```
Stonesoupextra
```

Exploiting Inputs

Any value that contains characters not in a-zA-Z. For example:

```
"not valid"
```

```
"badpassw0rd"
```

Source Code

For more information, please refer to the weakness Source Code.

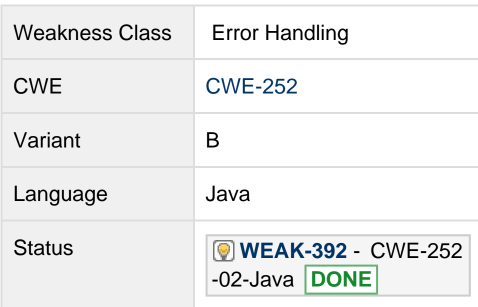
Related Information

- Execution Environment
- Restrictions and Requirements

J - CWE-252B - Unchecked Return

Summary

This snippet reads data from a sensitive file without checking the return value. It then repeats the operation on a non-sensitive file and prints out that file. If the first sensitive file is longer than the second non-sensitive file, then when the snippet prints to the non-sensitive file, it will print out data that was read from the longer sensitive file.



Weakness Class	Error Handling
CWE	CWE-252
Variant	B
Language	Java
Status	WEAK-392 - CWE-252-02-Java DONE

Negative Technical Impact(s)	Read Application Data

Implementation

The snippet creates a buffer and opens a two files. One sensitive and one non-sensitive.

```
final int STONESOUP_BUFFER_SIZE = 2048;
String stonesoup_sensitiveFName = stonesoup_ss_sensitive_file;
String stonesoup_otherFName = System.getenv("SS_OTHER_FILE");
byte[] stonesoup_buff = new byte[STONESOUP_BUFFER_SIZE];

java.io.InputStream stonesoup_sensitiveFile = new java.io.FileInputStream(
     stonesoup_sensitiveFName);
java.io.InputStream stonesoup_otherFile = new java.io.FileInputStream(
     stonesoup_otherFName);
```

The snippet then reads the sensitive file into the buffer.

```
/* STONESOUP: CROSSOVER POINT */
stonesoup_sensitiveFile.read(stonesoup_buff);
stonesoup_sensitiveFile.close();
```

It then reads the non-sensitive file into the same buffer.

```
/* STONESOUP: TRIGGER POINT */
stonesoup_otherFile.read(stonesoup_buff);
stonesoup_otherFile.close();
```

The buffer is then print out, and if the non-sensitive file was shorter than the sensitive one, then the end of the buffer will contain data from the sensitive file.

```
String output_data = new String(stonesoup_buff);
output.println("Output is:\n" + output_data);
```

Benign Inputs

Any file that is longer than the sensitive file. For example:

```
Sensitive file contents: root:x:0:0:root:/root:/bin/bash
Non-sensitive file contents: "The quick brown fox jumps over the lazy dog."
```

```
Sensitive file contents: sshd:x:105:65534::/var/run/sshd:/usr/sbin/nologin
Non-sensitive file contents: "Welcome to Linux Mint 14 Nadia (GNU/Linux
3.5.0-28-generic x86_64)\n\nWelcome to Linux Mint\n* Documentation:
http://www.linuxmint.com\n"
```

Exploiting Inputs

Any file that is shorter than the sensitive file. For example:

```
Sensitive file contents:
"root:x:0:0:root:/root:/bin/bash\ndaemon:x:1:1:daemon:/usr/sbin:/bin/sh\nbin:x:2:2:b
in:/bin:/bin/sh\nsys:x:3:3:sys:/dev:/bin/sh\nsync:......"
Non-sensitive file contents: "foo\n"
```

```
Sensitive file contents:
"root:x:0:\ndaemon:x:1:\nbin:x:2:\nsys:x:3:\nadm:x:4:someguy\ntty:x:5:\ndisk:x:6:\nl
p:x:7:\nmail:x:8:\nnews:x:9:\nuucp:x:10:\nman:x:12:\nproxy:x:13:\nkmem:x:15:\ndialou
t:x:20:\nfax......"
Non-sensitive file contents: "\n"
```

Source Code

For more information, please refer to the weakness Source Code.

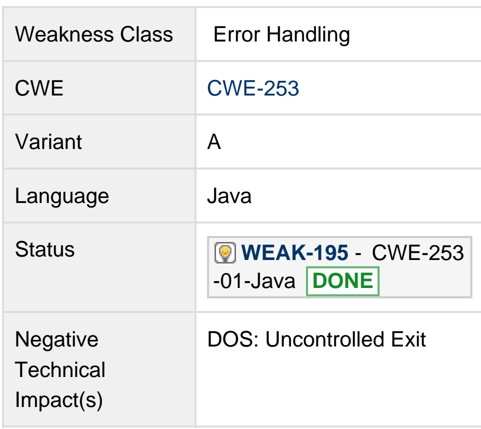
Related Information

- Execution Environment
- Restrictions and Requirements

J - CWE-253A - Incorrect Check Of Function Return Value

Summary

This snippet finds the first index of the character '.' in an input string. It then uses that index to obtain a sub-string of the input string. If the '.' character is not in the input string, then the index will be 1 and the sub-string operation will fail.



Weakness Class	Error Handling
CWE	CWE-253
Variant	A
Language	Java
Status	WEAK-195 - CWE-253-01-Java DONE
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

The snippet finds the index of the character '.'

```
int location = stonesoup_tainted_buff.indexOf('.');
```

The index location is then used in an attempt to get a sub-string of the original input. If the character '.' does not appear in the input, then the index will be -1 and the program will throw an exception.

```
/* STONESOUP: CROSSOVER POINT */
if (location != 0) {
    String substring;
    try {
         /* STONESOUP: TRIGGER POINT */
         substring = stonesoup_tainted_buff.substring(location);
    } catch (RuntimeException e) {
         e.printStackTrace(output);
         throw e;
    }
    output.println("Substring beginning with '.' is \"" + substring + "\"\n");
} else {
    output.println("'.' appears at start of line\n");
}
```

Benign Inputs

Any value that contains the character '.'. For example:

```
"The quick brown fox jumps over the lazy dog."
```

```
"25.69"
```

```
".14159"
```

Exploiting Inputs

Any value that doesn't contain the character '.'. For example:

```
"314159"
```

```
"All good boys deserve fruit"
```

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Restrictions and Requirements

J - CWE-390A - Detection of Error Condition Without Action

Summary

This snippet takes a file name and attempts to read it and output its contents to the screen. If the file name doesn't exist, it swallows a FileNotFoundException and later dies on a NullPointerException.

This weakness differs from 391 in that an exception is caught but nothing is done. As opposed to 391 which catches the exception only for the purpose of logging that it happened.

Weakness Class	Error Handling
CWE	CWE-390
Variant	A
Language	Java
Status	**WEAK-209** - CWE-390 -01-Java DONE
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

The snippet attempts to open a file. If the file doesn't exist, and exception is thrown, but nothing happens and the program continues on as if a file had been found.

```
java.io.FileInputStream fis = null;
java.io.File readPath = new java.io.File(filename);

/* STONESOUP: CROSSOVER POINT (Detection of Error Condition Without Action) */
try {
     fis = new java.io.FileInputStream(readPath);
} catch (java.io.FileNotFoundException e) {
}
```

The snippet then attempts to read from the file and output it to the screen. If no file is found, this results in a NullPointerException caught by a generic RuntimeException for scoring.

```
java.io.BufferedReader reader = new java.io.BufferedReader(new
java.io.InputStreamReader(fis));
String line = null;
try {
     while ((line = reader.readLine()) != null) {
          output.println(line);
     }
} catch (IOException e) {
     output.printf("Failed to read file.\n");
}
```

Benign Inputs

Any value that contains a filename that exists. For example:

```
file.txt
contents of file: "The quick brown fox jumps over the lazy dog."
```

```
anotherfile.txt
contents of file: "some text goes here"
```

```
thisfileexists.txt
contents of file: "non-empty"
```

Exploiting Inputs

Any value that contains a filename that doesn't. For example:

```
doesnotexist.txt
```

```
missingfile.txt
```

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Restrictions and Requirements

J - CWE-391A - Unchecked Error Condition

Summary

This snippet takes an integer for input to use as the size of an array. If the input is large enough, then the array will fail to be allocated. If this happens, an exception is thrown and caught but no action is taken.

This weakness differs from 390 in that an exception is only caught for the purposes of logging that it happened. As opposed to 390 which catches the exception, and just doesn't do anything with/about it.

Weakness Class	Error Handling
CWE	CWE-391
Variant	A
Language	Java
Status	WEAK-211 - CWE-391-01-Java DONE
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

The snippet takes an integer to be used as the size of an array.

```
int [] stonesoup_arr = null;

try {
     output.printf("Allocating array of size %d\n", size);
     stonesoup_arr = new int[size];        // attempt to allocate array of size
'size'
} catch (OutOfMemoryError e) {                        // catch an exception but do
nothing
     Tracer.tracepointError(e.getClass().getName() + ": " + e.getMessage());
     /* STONESOUP: CROSSOVER POINT (unchecked error condition) */
}
```

The snippet then attempts to use the array. If it wasn't allocated, then attempting to use it will result in an error caught by a generic RuntimeException.

```
for (int i = 0; i < stonesoup_arr.length; i++) {
     stonesoup_arr[i] = size - i;    // doesn't have space to allocate on heap,
errors out
}
```

Benign Inputs

Any value that doesn't cause the array to be unallocated due to memory constraints. For example:

```
1
```

```
561
```

```
9001
```

Exploiting Inputs

Any value that causes the array to be unallocated due to memory constraints. For example:

```
987654321
```

```
668554223
```

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Restrictions and Requirements

J - CWE-460A - Improper Cleanup On Thrown Exception

Summary

This snippet takes an integer for input to use as the size of an array. If the input is large enough, then an exception is raised and the array is initialized to a default size without ever changing the initial size variable that continues to be used. This leads to an ArrayIndexoutOfBoundsException when the array is accessed.

Weakness Class	Error Handling
CWE	CWE-460
Variant	A
Language	Java
Status	**WEAK-217** - CWE-460-01-Java **DONE**
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

The snippet takes an integer to be used as the size of an array. If the input is too large, this will result in an exception. After catching the exception, the snippet creates a default sized array instead.

```
int [] stonesoup_arr = null;

try {
     output.printf("Allocating array of size %d\n", size);
     stonesoup_arr = new int[size];
} catch (java.lang.OutOfMemoryError e) {
     /* STONESOUP: crossover point (improper cleanup) */
     stonesoup_arr = new int[100];
}
```

The snippet then uses the array. If the default array was created due to an exception, Then the initial input that is used as the size to access the array will cause an error caught by a generic RuntimeException

```
int i = size -1;
do {
     // do stuff that includes incorrectly cleaned up data
     stonesoup_arr[i--] = i;
} while (i > 0);
```

Benign Inputs

Any value that doesn't cause the array to be unallocated due to memory constraints. For example:

```
100
```

```
1324
```

```
9001
```

Exploiting Inputs

Any value that causes the array to be unallocated due to memory constraints. For example:

```
2000000000
```

```
1900000001
```

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Restrictions and Requirements

J - CWE-584A - Return Inside Finally

Summary

This snippet takes a file name as input. It attempts to get the absolute path using a created class GetAbsolutePath. The class has a method that returns inside a finally. If this is triggered by a file not existing, then it will result in any warnings being dropped and eventually a NullPointerException.

Weakness Class	Error Handling
CWE	CWE-584
Variant	A
Language	Java
Status	WEAK-218 - CWE-584 -01-Java DONE
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

The snippet takes a file name and attempts to get the absolute path of the filename.

```
File file;
GetAbsolutePath getpath = new GetAbsolutePath(taintedValue, output);

try {
     absPath = getpath.getAbsolutePath();
     validPath = true;
} catch (InvalidPathException e) {
     output.println("STONESOUP: Absolute path to file was not found.");
}
```

The GetAbsolutePath class has a function that returns inside a finally. This can happen if you provide a file name for a file that doesn't exist.

```
@SuppressWarnings("finally")
public String getAbsolutePath() throws InvalidPathException {
     String absName = null;
     try {
          absName = this.verifyAbsolutePath();
     } catch (InvalidPathException e) {
          output.println("STONESOUP: Error in verifying absolute path\n");
          throw e;
     } finally {
          /* STONESOUP: CROSSOVER POINT (return inside finally) */
          return absName;
     }
}
```

After the getAbsolutePath function returns, the program continues on as if a path has been found. This will happen if an exception was thrown or not because of the return being inside the finally block. When the rest of the program executes and attempts to open the file, there won't be any path to open, and this will result in an error.

```
if (validPath) {
     try {
          /* STONESOUP: TRIGGER POINT (return inside finally) */
          file = new File(absPath);
          freader = new Scanner(file);

          while (freader.hasNextLine()) {
               output.println(freader.nextLine());
          }
     } catch (NullPointerException e) {
          e.printStackTrace(output);
          throw e;
     } catch (FileNotFoundException e) {
          output.println("STONESOUP: File not found.");
     }
}
```

Benign Inputs

Any value that is a valid file that exists. For example:

```
file: i_exist_yo.txt
contents of file: "This is some text, it is in the file."
```

```
file: aFile.txt
contents of file: "One Fish, Two Fish, Red fish, Blue Fish."
```

```
file: asdfasdfasdf.txt
contents of file: "asdfasdfasdf\nasdfasdfasdf\nasdfasdfasdf\n"
```

Exploiting Inputs

Any value that is not a valid file that doesn't exist. For example:

```
this_file_does_not_exist_kthanksbai.txt
```

```
whatareyouevenlookingfor.txt
```

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Restrictions and Requirements

Memory Corruption

The memory corruption weakness class pertains to code that performs operations on a memory buffer in an unintended way, leading to abuse or corruption of the intended program control flow. It includes buffer overflows and underflows, out-of-bounds accesses, and memory safety errors. Often these issues arise when access is mistakenly granted to a memory location that is not intended. In general, this class contains issues where the developer incorrectly handles a buffer leading to overflow/underflow, corruption, or unintended access.

Weakness Members

The memory corruption weakness class is defined as a set of weakness types as defined by the MITRE Common Weakness Enumeration (CWE) ontology. The MITRE CWE ontology is defined a a tree, and as such may target CWEs that are a parent or child of another CWE. In those cases, CWEs may be proposed to remove from the test set target given that they are fully covered by one or more other CWEs. Individual weakness variants (i.e. tests) are designed around the definition, description, and code examples provided for each CWE.

CWE	Name	Target Source Languages
CWE-120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')	C
CWE-124	Buffer Underwrite ('Buffer Underflow')	C
CWE-126	Buffer Over-read	C
CWE-127	Buffer Under-read	C
CWE-129	Improper Validation of Array Index	C
CWE-134	Uncontrolled Format String	C
CWE-170	Improper Null Termination	C
CWE-415	Double Free	C
CWE-416	Use After Free	C
CWE-590	Free of Memory not on the Heap	C
CWE-761	Free of Pointer not at Start of Buffer	C
CWE-785	Use of Path Manipulation Function without Maximum-sized Buffer	C

CWE-805	Buffer Access with Incorrect Length Value	C
CWE-806	Buffer Access Using Size of Source Buffer	C
CWE-822	Untrusted Pointer Dereference	C
CWE-824	Access of Uninitialized Pointer	C
CWE-843	Access of Resource Using Incompatible Type ('Type Confusion')	C

Weakness Variants

The following weakness variants are developed and for the C and Java source languages.

C Source Language

Error rendering macro 'detailssummary' : In template Confluence.Templates.Macro.MasterDetail.contentLink2: When evaluating "contextPath()": Error while computing function "contextPath()": null

Java Source Language

The Java source language is not a target for memory corruption tests as the Java Virtual Machine (JVM) provides checked access memory and buffers.

Runtime Information

No specific runtime information is required for this weakness class. For information on the expected runtime environment, please refer to Execution Environment.

Notes

The design of the weakness algorithmic variants requires that an observable be generated to determine if the vulnerability was in fact triggered by exploiting input. Given that testing is to be run in an automated fashion, the point at which the weakness is triggered and the observable is generated must be deterministic. Unfortunately, a C program has limited control over memory layout, which makes generating reliable errors difficult. For instance, the C compiler provides no guarantees about the order of local variables within the stack frame, or even about which direction the stack grows. This means that it is difficult to create exploits that reliably overwrite/underwrite particular local variables in the stack frame or particular pointers (such as the instruction pointer). It means that a simple overread/underread may access different data on the stack in different program runs. Similarly, the compiler provides no guarantees about which memory locations will be returned for heap requests. Repeated heap requests may yield adjacent data regions, or regions far apart from each other.

In spite of these limitations, it is imperative that the weaknesses maintain a deterministic failure point to support automated scoring. For most memory corruption weaknesses, we provide 4 variants:

- Stack buffer
- Heap buffer
- Stack struct containing a buffer
- Heap struct containing a buffer

For simple stack and heap allocated buffers, the exploiting inputs must abuse memory to a large extent to increase the probability that the intended negative technical impact will be trigger and observed. For simple stack and heap allocated buffers, we do not test reads or writes that go one byte (or only a few bytes) outside our allocated boundary, because we cannot reliably detect the inappropriate read/write.

When we allocate a buffer within a struct on either the stack or the heap, the layout of fields within the struct is guaranteed. This means that when we overwrite by one byte, we know exactly what data we are writing into and can arrange for the one-byte overwrite to have a reliable negative technical impact. When we overread by one byte, we know exactly what data we are reading into and can check if that data is returned. Because of this control, for struct variants we use some exploit inputs that are large and some that are minimal (as little as one byte). One caveat regarding this behavior: since the layout of fields within a struct is guaranteed, and since the weakness is accessing other fields within the struct using a pointer to that same struct, is this obviously a disallowed action? Or is this akin to using well-defined numerical overflow in a hashing algorithm?

C - CWE-120A - Stack Buffer Overflow

Summary

This snippet implements an unchecked write into a stack allocated buffer. The buffer is declared as a fixed size local variable within a function. Untrusted input is not properly sanitized or restricted before being copied into the target buffer, resulting in a buffer overflow. The overflow potentially modifies other local variables, depending on the layout of the stack. If the stack grows downward, the overflow will eventually overwrite %eip, smashing the stack or setting it to a valid instruction.

For weakness C-CWE-120A, the buffer overflow occurs on the stack. It causes a problem when it overflows into compiler-generated stack values such as %eip.

For weakness C-CWE-120B, the buffer overflow occurs on the heap. It causes a problem when it over-writes memory manager values or user-defined values on the heap.

For weakness C-CWE-120C, the buffer overflow occurs on the heap, but it first overflows within a user-defined struct. This allows the weakness to have full control of the error behavior.

For weakness C-CWE-120D, the buffer overflow occurs on the stack, but it first overflows within a user-defined struct. This allows the weakness to have full control of the error behavior.

These weakness differ from CWE124, because CWE124 is a buffer underflow and these are buffer overflows.

They differ from CWE126, because CWE126 is a buffer over-read, and these are buffer over-writes.

Weakness Class	Memory Corruption
CWE	CWE-120
Variant	A
Language	C
Status	**WEAK-111** - CWE-120-0-C **DONE**
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

The snippet allocates a character buffer of size 64 on the stack.

```
char stonesoup_stack_buffer_64[64];
```

The untrusted input character array is then copied directly to the buffer without any size checking. The weakness has now reached the crossover point, or the point at which the overflow has occurred leaving the program in an undefined and corrupt state.

```
/* STONESOUP: CROSSOVER-POINT (Unchecked buffer copy) */
strcpy(stonesoup_stack_buffer_64,STONESOUP_TAINT_SOURCE);
```

After this, assuming that the for loop counter has not been overwritten, the entire character buffer (size 64) is converted to capital letters. The unchecked copy into the stack buffer has now allowed for the stack to be smashed, overwriting elements of the stack frame, such as %eip, with the input data. The point at which the result of the buffer overflow is triggered is on the return of the containing function. At that point, the previous stack frame is restored and the address in %eip is used to identify the next instruction. The input provided could overwrite %eip with a valid or invalid address. In the case of a valid address, the input could contain the instructions to execute, assuming executable stack is enabled on the binary, or a more sophisticated attack could be created using Return-Oriented Programing (ROP) techniques, assuming enough ROP gadgets exist to create a functional payload.

A stack buffer overflow may also result in the modification of other declared stack variables, resulting in modifying the behavior of the function itself. Unfortunately, a C program has limited control over memory layout, which makes generating reliable errors difficult. For instance, the C compiler provides no guarantees about the order of local variables within the stack frame, or even about which direction the stack grows. As a result, exploiting inputs result in a large overflow in an attempt to ensure reaching %eip and smashing the stack.

Benign Inputs

Any string of 63 characters or less is a benign input. For example:

```
AAA
```

```
Hello World!
```

```
Good-bye cruel world!
```

Due to the fact that none of these inputs are greater than the buffer size of 64 characters (63 char and one null byte) they will not result in a bad state.

Exploiting Inputs

Any string of 64 characters or greater is an exploiting input because the strcpy() function will include the null terminator. However, a larger string is more likely to result in an observable bad state since an overflow by one is not guaranteed to overwrite any critical memory. For example:

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAA
```

```
This is probably going to overflow a buffer of some sort.  Let's run it anyway and
see what happens.  I hope that everything turns out ok, because I would love to see
what this looks like in all CAPS.  Converting ASCII characters to upper case is
easy, you can either call a system function, or just add the standard difference
between the start of the two character sets.  But, I am sure there is a reason that
system functions exist for this sort of functionality.
```

Since both of these strings are significantly greater than the buffer size of 64 characters (63 char and one null byte) they will overflow into random memory and cause an observable bad state.

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Memory Corruption
- Restrictions and Requirements

Approved for public release; distribution unlimited.

C - CWE-120B - Heap Buffer Overflow

Summary

This snippet implements an unchecked write into a heap allocated buffer. The buffer is malloc'ed with a fixed size. Untrusted input is not properly sanitized or restricted before being copied into the target buffer, resulting in a buffer overflow. The overflow potentially modifies other variables in the heap, including special values used by the memory manager to keep track of which memory is allocated and which has been freed. Depending upon the layout of the heap, the overflow will eventually overwrite one of these special values, resulting in a crash from the memory manager.

For weakness C-CWE-120A, the buffer overflow occurs on the stack. It causes a problem when it overflows into compiler-generated stack values such as %eip.

For weakness C-CWE-120B, the buffer overflow occurs on the heap. It causes a problem when it over-writes memory manager values or user-defined values on the heap.

For weakness C-CWE-120C, the buffer overflow occurs on the heap, but it first overflows within a user-defined struct. This allows the weakness to have full control of the error behavior.

For weakness C-CWE-120D, the buffer overflow occurs on the stack, but it first overflows within a user-defined struct. This allows the weakness to have full control of the error behavior.

These weakness differ from CWE124, because CWE124 is a buffer underflow and these are buffer overflows.

They differ from CWE126, because CWE126 is a buffer over-read, and these are buffer over-writes.

Weakness Class	Memory Corruption
CWE	CWE-120
Variant	B
Language	C
Status	WEAK-109 - CWE-120-1-C DONE
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

The snippet allocates a pointer to a char on the stack.

```
char *stonesoup_heap_buffer_64 = 0;
```

Memory is allocated for the buffer on the heap.

```
stonesoup_heap_buffer_64 = (char*) malloc(64 * sizeof(char));
```

The untrusted input character array is then copied directly to the buffer without any size checking. The weakness has now reached the crossover point, or the point at which the overflow has occurred leaving the program in an undefined and corrupt state. This point is also marked as the trigger point, because the failure may occur any time the heap is accessed after this point.

```
/* STONESOUP: CROSSOVER-POINT (Unchecked buffer copy) */
/* STONESOUP: TRIGGER-POINT (Buffer Overflow: Unchecked heap buffer copy) */
strcpy(stonesoup_heap_buffer_64, STONESOUP_TAINT_SOURCE);
```

After this, the entire character buffer (size 64) is converted to capital letters. The unchecked copy into the heap buffer has now allowed for the

heap to be smashed, overwriting elements of the memory management system and other heap values with the input data. The point at which the result of the buffer overflow is triggered is on some subsequent heap access, often when the buffer itself is freed. The over-write may alter function pointers that are stored in memory. If executable heap is enabled, this may cause execution of arbitrary code. Even if executable heap is disabled, over-writing function pointers may result in execution of code chosen by the attacker or may result in a crash if the address written is not valid for execution.

Unfortunately, a C program has limited control over the heap layout (unless the program runs its own memory manager). This means that it is difficult to generate reliable errors from heap buffer overflows. The standard memory manager provides no guarantees about the order of data allocated on the heap. To mitigate this issue, exploiting inputs result in a large overflow in an attempt to destroy something on the heap.

Benign Inputs

Any string of 63 characters or less is a benign input. For example:

```
AAA
```

```
Hellow World!
```

```
Good-bye cruel world!
```

Due to the fact that none of these inputs are greater than the buffer size of 64 characters (63 char and one null byte) they will not result in a bad state.

Exploiting Inputs

Any string of 64 characters or greater is an exploiting input because the strcpy() function will include the null terminator. However, a larger string is more likely to result in an observable bad state since an overflow by one is not guaranteed to overwrite any critical memory. For example:

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAA
```

```
This is probably going to overflow a buffer of some sort.  Let's run it anyway and
see what happens.  I hope that everything turns out ok, because I would love to see
what this looks like in all CAPS.  Converting ASCII characters to upper case is
easy, you can either call a system function, or just add the standard difference
between the start of the two character sets.  But, I am sure there is a reason that
system functions exist for this sort of functionality.
```

Since both of these strings are significantly greater than the buffer size of 64 characters (63 char and one null byte) they will overflow into random memory and cause an observable bad state.

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Restrictions and Requirements

C - CWE-120C - Heap Buffer Overflow within Struct

Summary

This snippet implements an unchecked write into a buffer is contained within a heap-allocated struct. The struct contains a function pointer, a fixed-size buffer, and another function pointer. Untrusted input is not properly sanitized or restricted before being copied into the target buffer, resulting in a buffer overflow. The overflow will destroy one of the function pointers either side of the buffer. The weakness then accesses those pointers, resulting in a crash or execution of attacker-specified code.

For weakness C-CWE-120A, the buffer overflow occurs on the stack. It causes a problem when it overflows into compiler-generated stack values such as %eip.

For weakness C-CWE-120B, the buffer overflow occurs on the heap. It causes a problem when it over-writes memory manager values or user-defined values on the heap.

For weakness C-CWE-120C, the buffer overflow occurs on the heap, but it first overflows within a user-defined struct. This allows the weakness to have full control of the error behavior.

For weakness C-CWE-120D, the buffer overflow occurs on the stack, but it first overflows within a user-defined struct. This allows the weakness to have full control of the error behavior.

These weakness differ from CWE124, because CWE124 is a buffer underflow and these are buffer overflows.

They differ from CWE126, because CWE126 is a buffer over-read, and these are buffer over-writes.

Weakness Class	Memory Corruption
CWE	CWE-120
Variant	C
Language	C
Status	**WEAK-399** - CWE-120-2-C **DONE**
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

The snippet defines struct stonesoup_struct, with a function pointer called before, a 64-character buffer, and function pointer called after.

```
struct stonesoup_struct {
    int (* before)(int);
    char buffer[64];
    int (* after)(int);
};
```

It creates a pointer to a struct stonesoup_struct on the stack.

```
struct stonesoup_struct* stonesoup_data = NULL;
```

Memory is allocated for the struct on the heap.

```
stonesoup_data = (struct stonesoup_struct*) malloc(sizeof(struct stonesoup_struct));
```

The before and after points are set to a function that converts a single character to upper case, and the buffer itself is zeroed.

```
stonesoup_data->before = stonesoup_toupper;
memset(stonesoup_data->buffer,0,64);
stonesoup_data->after = stonesoup_toupper;
```

The untrusted input character array is then copied directly to the buffer inside the struct without any size checking. The weakness has now reached the crossover point, or the point at which the overflow has occurred leaving the program in an undefined and corrupt state. This point is also marked as the trigger point, because the failure may occur any time the heap is accessed after this point.

```
strcpy(stonesoup_data->buffer, STONESOUP_TAINT_SOURCE);
```

After this, the entire character buffer (size 64) is converted to capital letters one-by-one using the after pointer from the struct.

```
stonesoup_printf("%c",stonesoup_data->after(stonesoup_data->buffer[stonesoup_i]));
```

Since the after pointer has been over-written, this will cause unpredictable behavior. For our particular exploit values, this will cause a program crash because the program tries to access an invalid pointer. Depending on the input value, it may alternately allow execution of code chosen by the attacker. If executable heap is enabled, this may cause execution of arbitrary code.

Benign Inputs

Any string of 63 characters or less is a benign input. For example:

```
AAA
```

```
Hellow World!
```

```
Good-bye cruel world!
```

Due to the fact that none of these inputs are greater than the buffer size of 64 characters (63 char and one null byte) they will not result in a bad state.

Exploiting Inputs

Any string of 64 characters or greater is an exploiting input because the strcpy() function will include the null terminator. However, a larger string is more likely to result in an observable bad state since an overflow by one is not guaranteed to overwrite any critical memory. For example:

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAA
```

```
This is probably going to overflow a buffer of some sort.  Let's run it anyway and
see what happens.  I hope that everything turns out ok, because I would love to see
what this looks like in all CAPS.  Converting ASCII characters to upper case is
easy, you can either call a system function, or just add the standard difference
between the start of the two character sets.  But, I am sure there is a reason that
system functions exist for this sort of functionality.
```

Since both of these strings are significantly greater than the buffer size of 64 characters (63 char and one null byte) they will overflow into the after pointer and cause a system crash.

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Restrictions and Requirements

C - CWE-120D - Stack Buffer Overflow within Struct

Summary

This snippet implements an unchecked write into a buffer that is contained within stack-allocated struct. The struct contains a function pointer, a fixed-size buffer, and another function pointer. Untrusted input is not properly sanitized or restricted before being copied into the target buffer, resulting in a buffer overflow. The overflow will destroy one of the function pointers either side of the buffer. The weakness then accesses those pointers, resulting in a crash or execution of attacker-specified code.

For weakness C-CWE-120A, the buffer overflow occurs on the stack. It causes a problem when it overflows into compiler-generated stack values such as %eip.

For weakness C-CWE-120B, the buffer overflow occurs on the heap. It causes a problem when it over-writes memory manager values or user-defined values on the heap.

For weakness C-CWE-120C, the buffer overflow occurs on the heap, but it first overflows within a user-defined struct. This allows the weakness to have full control of the error behavior.

For weakness C-CWE-120D, the buffer overflow occurs on the stack, but it first overflows within a user-defined struct. This allows the weakness to have full control of the error behavior.

These weakness differ from CWE124, because CWE124 is a buffer underflow and these are buffer overflows.

They differ from CWE126, because CWE126 is a buffer over-read, and these are buffer over-writes.

Weakness Class	Memory Corruption
CWE	CWE-120

Variant	D
Language	C
Status	WEAK-419 - CWE-120-3-C DONE
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

The snippet defines struct stonesoup_struct, with a function pointer called before, a 64-character buffer, and function pointer called after.

```
struct stonesoup_struct {
    int (* before)(int);
    char buffer[64];
    int (* after)(int);
};
```

It allocates a struct stonesoup_struct on the stack.

```
struct stonesoup_struct stonesoup_data;
```

The before and after points are set to a function that converts a single character to upper case, and the buffer itself is zeroed.

```
stonesoup_data.before = stonesoup_toupper;
for (stonesoup_i = 0; stonesoup_i < 64; stonesoup_i++) {
    stonesoup_data.buffer[stonesoup_i] = 0;
}
stonesoup_data.after = stonesoup_toupper;
```

The untrusted input character array is then copied directly to the buffer inside the struct without any size checking. The weakness has now reached the crossover point, or the point at which the overflow has occurred leaving the program in an undefined and corrupt state.

```
strcpy(stonesoup_data.buffer, STONESOUP_TAINT_SOURCE);
```

After this, the entire character buffer (size 64) is converted to capital letters one-by-one using the after pointer from the struct.

```
stonesoup_printf("%c",stonesoup_data.after(stonesoup_data.buffer[stonesoup_i]));
```

Since the after pointer has been over-written, this will cause unpredictable behavior. For our particular exploit values, this will cause a program crash because the program tries to access an invalid pointer. Depending on the input value, it may alternately allow execution of code chosen by the attacker. If executable stack is enabled, this may cause execution of arbitrary code.

Benign Inputs

Any string of 63 characters or less is a benign input. For example:

```
AAA
```

```
Hellow World!
```

```
Good-bye cruel world!
```

Due to the fact that none of these inputs are greater than the buffer size of 64 characters (63 char and one null byte) they will not result in a bad state.

Exploiting Inputs

Any string of 64 characters or greater is an exploiting input because the strcpy() function will include the null terminator. However, a larger string is more likely to result in an observable bad state since an overflow by one is not guaranteed to overwrite any critical memory. For example:

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAA
```

```
This is probably going to overflow a buffer of some sort.  Let's run it anyway and
see what happens.  I hope that everything turns out ok, because I would love to see
what this looks like in all CAPS.  Converting ASCII characters to upper case is
easy, you can either call a system function, or just add the standard difference
between the start of the two character sets.  But, I am sure there is a reason that
system functions exist for this sort of functionality.
```

Since both of these strings are significantly greater than the buffer size of 64 characters (63 char and one null byte) they will overflow into the after pointer and cause a system crash.

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Restrictions and Requirements

C - CWE-124A - Buffer Underwrite ('Buffer Underflow')

Summary

This snippet implements an incorrectly checked write into a stack allocated buffer. The buffer is declared as a fixed size local variable within a function. Untrusted input is not properly sanitized or restricted before being copied into the buffer, from the last character to the first. This allows input greater than 63 characters in length to underflow the buffer, overwriting arbitrary memory, and potentially modifying other local variables depending on the layout of the stack.

For weakness C-CWE-124A, the buffer underflow occurs on the stack. It causes a problem when it overflows into compiler-generated stack values or user-defined values on the stack.

For weakness C-CWE-124B, the buffer underflow occurs on the heap. It causes a problem when it over-writes memory manager values or user-defined values on the heap.

For weakness C-CWE-124C, the buffer underflow occurs on the heap, but it first underflow within a user-defined struct. This allows the weakness to have full control of the error behavior.

For weakness C-CWE-124D, the buffer underflow occurs on the stack, but it first underflow within a user-defined struct. This allows the weakness to have full control of the error behavior.

These weaknesses differ from CWE120, because CWE120 is a buffer overflow and these are buffer underflow.

These weaknesses differ from CWE127, because CWE127 is a buffer under-read, and these are buffer under-writes.

Weakness Class	Memory Corruption
CWE	CWE-124
Variant	A
Language	C
Status	**WEAK-118** - CWE-124-0-C **DONE**
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

The snippet allocates a character buffer of size 64 and a char* buffer of size 8 on the stack. The snippet also declares two integers, one containing the size of the newly allocated buffer (accounting for string null termination) and the other to contain the size of the untrusted input string.

```
char stonesoup_stack_buff_64[64];
char *stonesoup_other_buff[8];
int stonesoup_my_buff_size = 63;
int stonesoup_buff_size;
```

The snippet then sets the char* buffer's last element equal to a pointer to the input string and the input string's size is stored in the variable declared above. This size is used as the loop counter to copy each character of the input string, from the last to the first, into the internal buffer. Since the size checking is done with the size of the input string if the input string is greater than 63 characters in length the buffer will underflow, overwriting arbitrary memory.

```
stonesoup_other_buff[7] = STONESOUP_TAINT_SOURCE;
memset(stonesoup_stack_buff_64,65,64);
stonesoup_stack_buff_64[64 - 1] = '\0';
stonesoup_buff_size = ((int )(strlen(STONESOUP_TAINT_SOURCE)));

for (; stonesoup_buff_size >= 0; (--stonesoup_my_buff_size , --stonesoup_buff_size))
{
    /* STONESOUP: CROSSOVER-POINT (Buffer Underwrite) */
    /* STONESOUP: TRIGGER-POINT (Buffer Underwrite: Stack Underflow) */
    stonesoup_stack_buff_64[stonesoup_my_buff_size] =
STONESOUP_TAINT_SOURCE[stonesoup_buff_size];
}
```

Assuming that the snippet has not crashed, the entire character buffer (size 64) is then converted to capital letters. The unchecked copy into the stack buffer has now allowed for the stack to be smashed, overwriting elements of the stack frame with the input data. The point at which the result of the buffer overflow is triggered is on the call to get the size of the input string that is stored in 'stonesoup_other_buff'. Since the 'stonesoup_other_buff' is declared on the stack before 'stonesoup_stack_buff_64' (generally, however C does not have a guarantee as to memory layout) the underflow will overwrite the pointer to the input string and will cause `strlen()` to segfault due to an invalid pointer being passed to it.

Benign Inputs

Any string of 63 characters or less is a benign input. For example:

```
AAA
```

```
Hellow World!
```

```
Good-bye cruel world!
```

Due to the fact that none of these inputs are greater than the buffer size of 64 characters (63 char and one null byte) they will not result in a bad state.

Exploiting Inputs

Any string of 64 characters or greater is an exploiting input because the strcpy() function will include the null terminator. However, a larger string is more likely to result in an observable bad state since an overflow by one is not guaranteed to overwrite any critical memory. For example:

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAA
```

```
This is probably going to overflow a buffer of some sort.  Let's run it anyway and
see what happens.  I hope that everything turns out ok, because I would love to see
what this looks like in all CAPS.  Converting ASCII characters to upper case is
easy, you can either call a system function, or just add the standard difference
between the start of the two character sets.  But, I am sure there is a reason that
system functions exist for this sort of functionality.
```

Since both of these strings are significantly greater than the buffer size of 64 characters (63 char and one null byte) they will overflow into random memory and cause an observable bad state.

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Memory Corruption
- Restrictions and Requirements

C - CWE-124B - Buffer Underwrite ('Buffer Underflow')

Summary

This snippet implements an incorrectly checked write into a heap allocated buffer. The buffer is `malloc()`'d with a fixed size and pointed to by a local variable. Untrusted input is not properly sanitized or restricted before being copied into the buffer, from the last character to the first. This allows input greater than 63 characters in length to underflow the buffer, overwriting arbitrary memory, and potentially modifying other heap allocated variables depending on the layout of the heap.

For weakness C-CWE-124A, the buffer underflow occurs on the stack. It causes a problem when it overflows into compiler-generated stack values or user-defined values on the stack.

For weakness C-CWE-124B, the buffer underflow occurs on the heap. It causes a problem when it over-writes memory manager values or user-defined values on the heap.

For weakness C-CWE-124C, the buffer underflow occurs on the heap, but it first underflow within a user-defined struct. This allows the weakness to have full control of the error behavior.

For weakness C-CWE-124D, the buffer underflow occurs on the stack, but it first underflow within a user-defined struct. This allows the weakness to have full control of the error behavior.

These weaknesses differ from CWE120, because CWE120 is a buffer overflow and these are buffer underflow.

These weaknesses differ from CWE127, because CWE127 is a buffer under-read, and these are buffer under-writes.

Weakness Class	Memory Corruption
CWE	CWE-124
Variant	A
Language	C
Status	**WEAK-120** - CWE-124-1-C DONE

Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

The snippet allocates a character buffer of size 64 on the heap and declares two integers, one containing the size of the newly allocated buffer (accounting for string null termination) and the other to contain the size of the untrusted input string.

```
int stonesoup_i = 0;
int stonesoup_buff_size = 63;
int stonesoup_taint_len;
char *stonesoup_heap_buff_64 = 0;

stonesoup_heap_buff_64 = (char*) malloc(64 * sizeof(char));
```

The snippet then stores the input string's size in the variable declared above. This size is used as the loop counter to copy each character of the input string, from the last to the first, into the internal buffer. Since the size checking is done with the size of the input string if the input string is greater than 63 characters in length the buffer will underflow, overwriting arbitrary memory on the heap. This is both the crossover and the trigger point.

```
for (; stonesoup_taint_len >= 0; (--stonesoup_buff_size , --stonesoup_taint_len)) {
    /* STONESOUP: CROSSOVER-POINT (Buffer Underwrite) */
    /* STONESOUP: TRIGGER-POINT (Buffer Underwrite: Heap Underflow) */
    stonesoup_heap_buff_64[stonesoup_buff_size] =
STONESOUP_TAINT_SOURCE[stonesoup_taint_len];
}
```

Assuming that the snippet has not crashed, the entire character buffer (size 64) is then converted to capital letters. The unchecked copy into the heap buffer has now allowed for the heap to be smashed, overwriting elements on the heap with the input data. The point at which the result of the buffer overflow is triggered is on the call to free() at the end of the snippet. Since input data underflowed the buffer it has corrupted data stored before the buffer that is needed by `free()` to properly release the memory, causing a segfault.

```
free(stonesoup_heap_buff_64);
```

Benign Inputs

Any string of 63 characters or less is a benign input. For example:

```
AAA
```

```
Hellow World!
```

```
Good-bye cruel world!
```

Due to the fact that none of these inputs are greater than the buffer size of 64 characters (63 char and one null byte) they will not result in a bad state.

Exploiting Inputs

Any string of 64 characters or greater is an exploiting input because the strcpy() function will include the null terminator. However, a larger string is more likely to result in an observable bad state since an overflow by one is not guaranteed to overwrite any critical memory. For example:

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAA
```

```
This is probably going to overflow a buffer of some sort.  Let's run it anyway and
see what happens.  I hope that everything turns out ok, because I would love to see
what this looks like in all CAPS.  Converting ASCII characters to upper case is
easy, you can either call a system function, or just add the standard difference
between the start of the two character sets.  But, I am sure there is a reason that
system functions exist for this sort of functionality.
```

Since both of these strings are significantly greater than the buffer size of 64 characters (63 char and one null byte) they will overflow into random memory and cause an observable bad state.

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Memory Corruption
- Restrictions and Requirements

C - CWE-124C - Buffer Underwrite ('Buffer Underflow')

Summary

This snippet implements an incorrectly checked write into a buffer that is contained within a heap allocated struct. The buffer is declared as a struct member with a fixed size. Untrusted input is not properly sanitized or restricted before being copied into the buffer, from the last character to the first. This allows input greater than 63 characters in length to underflow the buffer, overwriting the function pointer declared before the buffer in the struct.

For weakness C-CWE-124A, the buffer underflow occurs on the stack. It causes a problem when it overflows into compiler-generated stack values or user-defined values on the stack.

For weakness C-CWE-124B, the buffer underflow occurs on the heap. It causes a problem when it over-writes memory manager values or user-defined values on the heap.

For weakness C-CWE-124C, the buffer underflow occurs on the heap, but it first underflow within a user-defined struct. This allows the weakness to have full control of the error behavior.

For weakness C-CWE-124D, the buffer underflow occurs on the stack, but it first underflow within a user-defined struct. This allows the weakness to have full control of the error behavior.

These weaknesses differ from CWE120, because CWE120 is a buffer overflow and these are buffer underflow.

These weaknesses differ from CWE127, because CWE127 is a buffer under-read, and these are buffer under-writes.

Weakness Class	Memory Corruption
CWE	CWE-124
Variant	A
Language	C
Status	WEAK-400 - CWE-124-2-C DONE
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

The snippet declares a character buffer of size 64 as a member of a struct allocated on the heap and declares two integers, one containing the size of the newly allocated buffer (accounting for string null termination) and the other to contain the size of the untrusted input string.

```
int stonesoup_i = 0;
struct stonesoup_struct* stonesoup_data = NULL;
int stonesoup_buff_size = 63;
int stonesoup_taint_len;

stonesoup_data = (struct stonesoup_struct*) malloc(sizeof(struct stonesoup_struct));
```

The snippet then sets the struct's 'before' and 'after' pointers equal to the '`stonesoup_toUpper()`' function and stores the input string's size in the variable declared above. This size is used as the loop counter to copy each character of the input string, from the last to the first, into the internal buffer. Since the size checking is done with the size of the input string if the input string is greater than 63 characters in length the buffer will underflow, overwriting the struct's 'before' pointer. This is both the crossover and the trigger point.

```
if (stonesoup_data != NULL) {
    stonesoup_data->before = stonesoup_toupper;
    memset(stonesoup_data->buffer, 'A', 64);
    stonesoup_data->buffer[63] = '\0';
    stonesoup_data->after = stonesoup_toupper;

    stonesoup_taint_len = ((int )(strlen(STONESOUP_TAINT_SOURCE)));
    for (; stonesoup_taint_len >= 0; (--stonesoup_buff_size ,
--stonesoup_taint_len)) {
        /* STONESOUP: CROSSOVER-POINT (Buffer Underwrite) */
        /* STONESOUP: TRIGGER-POINT (Buffer Underwrite: Heap Underflow) */
        stonesoup_heap_buff_64[stonesoup_buff_size] =
STONESOUP_TAINT_SOURCE[stonesoup_taint_len];
    }
```

Assuming that the snippet has not crashed, the entire character buffer (size 64) is then converted to capital letters. The unchecked copy into the buffer has now allowed for the heap to be smashed, overwriting elements on the heap with the input data, and more specifically, overwriting the 'before' pointer contained in the struct. The point at which the result of the buffer overflow is triggered is on the call to the struct's 'before' pointer. Since input data underflowed the buffer it has corrupted the function pointer stored in 'before' and calling it will result in a segfault.

```
for (stonesoup_i = 0; stonesoup_i < 64; ++stonesoup_i) {
    stonesoup_data->buffer[stonesoup_i] =
stonesoup_toupper(stonesoup_data->buffer[stonesoup_i]);

stonesoup_printf("%c",stonesoup_data->before(stonesoup_data->buffer[stonesoup_i]));
}
```

Benign Inputs

Any string of 63 characters or less is a benign input. For example:

```
AAA
```

```
Hellow World!
```

```
Good-bye cruel world!
```

Due to the fact that none of these inputs are greater than the buffer size of 64 characters (63 char and one null byte) they will not result in a bad state.

Exploiting Inputs

Any string of 64 characters or greater is an exploiting input because the strcpy() function will include the null terminator. However, a larger string is more likely to result in an observable bad state since an overflow by one is not guaranteed to overwrite any critical memory. For example:

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAA
```

```
This is probably going to overflow a buffer of some sort.  Let's run it anyway and
see what happens.  I hope that everything turns out ok, because I would love to see
what this looks like in all CAPS.  Converting ASCII characters to upper case is
easy, you can either call a system function, or just add the standard difference
between the start of the two character sets.  But, I am sure there is a reason that
system functions exist for this sort of functionality.
```

Since both of these strings are significantly greater than the buffer size of 64 characters (63 char and one null byte) they will overflow into random memory and cause an observable bad state.

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Memory Corruption
- Restrictions and Requirements

C - CWE-124D - Buffer Underwrite ('Buffer Underflow')

Summary

This snippet implements an incorrectly checked write into a buffer that is contained within a stack allocated struct. The buffer is declared as a struct member with a fixed size. Untrusted input is not properly sanitized or restricted before being copied into the buffer, from the last character to the first. This allows input greater than 63 characters in length to underflow the buffer, overwriting the function pointer declared before the buffer in the struct.

For weakness C-CWE-124A, the buffer underflow occurs on the stack. It causes a problem when it overflows into compiler-generated stack values or user-defined values on the stack.

For weakness C-CWE-124B, the buffer underflow occurs on the heap. It causes a problem when it over-writes memory manager values or user-defined values on the heap.

For weakness C-CWE-124C, the buffer underflow occurs on the heap, but it first underflow within a user-defined struct. This allows the weakness to have full control of the error behavior.

For weakness C-CWE-124D, the buffer underflow occurs on the stack, but it first underflow within a user-defined struct. This allows the weakness to have full control of the error behavior.

These weaknesses differ from CWE120, because CWE120 is a buffer overflow and these are buffer underflow.

These weaknesses differ from CWE127, because CWE127 is a buffer under-read, and these are buffer under-writes.

Weakness Class	Memory Corruption
CWE	CWE-124
Variant	A
Language	C
Status	**WEAK-420** - CWE-124-3-C DONE
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

The snippet declares a character buffer of size 64 as a member of a struct allocated on the stack and declares two integers, one containing the size of the newly allocated buffer (accounting for string null termination) and the other to contain the size of the untrusted input string.

```
int stonesoup_i = 0;
struct stonesoup_struct stonesoup_data;
int stonesoup_buff_size = 63;
int stonesoup_taint_len;
```

The snippet then sets the struct's 'before' and 'after' pointers equal to the '`stonesoup_toUpper()`' function and stores the input string's size in the variable declared above. This size is used as the loop counter to copy each character of the input string, from the last to the first, into the internal buffer. Since the size checking is done with the size of the input string if the input string is greater than 63 characters in length the buffer will underflow, overwriting the struct's 'before' pointer. This is both the crossover and the trigger point.

```
stonesoup_data.before = stonesoup_toupper;
for (stonesoup_i = 0; stonesoup_i < 64; stonesoup_i++) {
    stonesoup_data.buffer[stonesoup_i] = 'A';
}
stonesoup_data.buffer[63] = '\0';
stonesoup_data.after = stonesoup_toupper;


stonesoup_taint_len = ((int )(strlen(STONESOUP_TAINT_SOURCE)));
/* STONESOUP: CROSSOVER-POINT (Buffer Underwrite) */
/* STONESOUP: TRIGGER-POINT (Buffer Underwrite: Heap Underflow) */
for (; stonesoup_taint_len >= 0; (--stonesoup_buff_size , --stonesoup_taint_len)) {
    stonesoup_data.buffer[stonesoup_buff_size] =
STONESOUP_TAINT_SOURCE[stonesoup_taint_len];
}
```

Assuming that the snippet has not crashed, the entire character buffer (size 64) is then converted to capital letters. The unchecked copy into the buffer has now allowed for the stack to be smashed, overwriting elements on the stack with the input data, and more specifically, overwriting the 'before' pointer contained in the struct. The point at which the result of the buffer overflow is triggered is on the call to the struct's 'before' pointer. Since input data underflowed the buffer it has corrupted the function pointer stored in 'before' and calling it will result in a segfault.

```
for (stonesoup_i = 0; stonesoup_i < 64; ++stonesoup_i) {
    stonesoup_data.buffer[stonesoup_i] =
stonesoup_toupper(stonesoup_data.buffer[stonesoup_i]);

stonesoup_printf("%c",stonesoup_data.before(stonesoup_data.buffer[stonesoup_i]));
}
```

Benign Inputs

Any string of 63 characters or less is a benign input. For example:

```
AAA
```

```
Hellow World!
```

```
Good-bye cruel world!
```

Due to the fact that none of these inputs are greater than the buffer size of 64 characters (63 char and one null byte) they will not result in a bad state.

Exploiting Inputs

Any string of 64 characters or greater is an exploiting input because the strcpy() function will include the null terminator. However, a larger string is more likely to result in an observable bad state since an overflow by one is not guaranteed to overwrite any critical memory. For example:

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAA
```

```
This is probably going to overflow a buffer of some sort.  Let's run it anyway and
see what happens.  I hope that everything turns out ok, because I would love to see
what this looks like in all CAPS.  Converting ASCII characters to upper case is
easy, you can either call a system function, or just add the standard difference
between the start of the two character sets.  But, I am sure there is a reason that
system functions exist for this sort of functionality.
```

Since both of these strings are significantly greater than the buffer size of 64 characters (63 char and one null byte) they will overflow into random memory and cause an observable bad state.

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Memory Corruption
- Restrictions and Requirements

C - CWE-126A - Buffer Over-Read

Summary

This snippet implements an unchecked read from a buffer. The buffer is declared as a fixed size buffer on the stack. Untrusted input is not properly sanitized or restricted before being used to determine the number of characters to read from the buffer. This allows input greater than 63 characters in length to over read the buffer, printing the contents of the memory located after the buffer on the stack.

For weakness C-CWE-126A, the buffer over-read occurs on the stack.

For weakness C-CWE-126B, the buffer over-read occurs on the heap.

For weakness C-CWE-126C, the buffer over-read occurs on the heap, printing an observable set within a user-defined struct.

For weakness C-CWE-126D, the buffer over-read occurs on the stack, printing an observable set within a user-defined struct.

These weaknesses differ from CWE120, because CWE120 is a buffer overwrite, and these are buffer over-reads.

These weaknesses differ from CWE127, because CWE127 is a buffer under-read, and these are buffer over-reads.

Weakness Class	Memory Corruption

CWE	CWE-126
Variant	A
Language	C
Status	WEAK-124 - CWE-126-0-C DONE
Negative Technical Impact(s)	Information Disclosure

Implementation

The snippet allocates a character buffer of size 64 on the stack and declares two integers, one to contain the size of the untrusted input string and the other set to 0 to be used as a loop counter later in execution.

```
int stonesoup_ss_i = 0;
char stonesoup_stack_buff_64[64];
int stonesoup_my_buff_size;
```

The snippet then fills the newly allocated buffer with 'A's and null terminates it. After the buffer has been filled, the snippet gets the size of the input string and saves it to the variable declared above. This size is then used to iterate through the buffer, printing each character to the output stream. Since the size is not sanitized before being used to iterate over the buffer, an input string of 65 characters or more will cause the snippet to unintentionally disclose the data stored in memory addresses after the buffer on the stack. This is both the crossover and the trigger point.

```
memset(stonesoup_stack_buff_64,'A',63);
stonesoup_stack_buff_64[63] = '\0';


stonesoup_my_buff_size = ((int )(strlen(STONESOUP_TAINT_SOURCE)));
for (; stonesoup_ss_i < stonesoup_my_buff_size; ++stonesoup_ss_i){
    /* STONESOUP: CROSSOVER-POINT (Buffer Overread) */
    /* STONESOUP: TRIGGER-POINT (Buffer Overread) */
    stonesoup_printf("%c",stonesoup_stack_buff_64[stonesoup_ss_i]);
}
```

Benign Inputs

Any string of 64 characters or less is a benign input. For example:

```
AAAAAAAAAAAAAAAAAAAAAAA
```

```
Hello World!
```

```
Short is good
```

Due to the fact that none of these inputs are greater than the buffer size of 64 characters (63 char and one null byte) they will not result in a

bad state.

Exploiting Inputs

Any string of 54 characters or greater is an exploiting input. For example:

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAA
```

```
This is probably going to overflow a buffer of some sort.  Let's run it anyway and
see what happens.  I hope that everything turns out ok, because I would love to see
what this looks like in all CAPS.  Converting ASCII characters to upper case is
easy, you can either call a system function, or just add the standard difference
between the start of the two character sets.  But, I am sure there is a reason that
system functions exist for this sort of functionality.
```

Since both of these strings are significantly greater than the buffer size of 64 characters (63 char and one null byte) they will overflow into random memory and cause an observable bad state.

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Memory Corruption
- Restrictions and Requirements

C - CWE-126B - Buffer Over-Read

Summary

This snippet implements an unchecked read from a buffer. The buffer is allocated as a fixed size buffer on the heap. Untrusted input is not properly sanitized or restricted before being used to determine the number of characters to read from the buffer. This allows input greater than 63 characters in length to over read the buffer, printing the contents of the memory located after the buffer on the heap.

For weakness C-CWE-126A, the buffer over-read occurs on the stack.

For weakness C-CWE-126B, the buffer over-read occurs on the heap.

For weakness C-CWE-126C, the buffer over-read occurs on the heap, printing an observable set within a user-defined struct.

For weakness C-CWE-126D, the buffer over-read occurs on the stack, printing an observable set within a user-defined struct.

These weaknesses differ from CWE120, because CWE120 is a buffer overwrite, and these are buffer over-reads.

These weaknesses differ from CWE127, because CWE127 is a buffer under-read, and these are buffer over-reads.

Weakness Class	Memory Corruption
CWE	CWE-126
Variant	A
Language	C
Status	WEAK-126 - CWE-126-1-C DONE
Negative Technical Impact(s)	Information Disclosure

Implementation

The snippet allocates a character buffer of size 64 on the heap and declares two integers, one to contain the size of the untrusted input string and the other set to 0 to be used as a loop counter later in execution.

```
int stonesoup_ss_i = 0;
char* stonesoup_heap_buff_64;
int stonesoup_buff_size;

stonesoup_heap_buff_64 = (char*) malloc(64 * sizeof(char));
```

The snippet then fills the newly allocated buffer with 'A's and null terminates it. After the buffer has been filled with 'A's, the snippet gets the size of the input string and saves it to the variable declared above. The snippet then safely copies the input string into the internal buffer. It then uses the size of the input string to iterate through the buffer, printing each character to the output stream. Since the size is not sanitized before being used to iterate over the buffer, an input string of 65 characters or more will cause the snippet to unintentionally disclose the data stored in memory addresses after the buffer on the heap. This is both the crossover and the trigger point.

```
memset(stonesoup_heap_buff_64,'A',63);
stonesoup_heap_buff_64[63] = '\0';

stonesoup_buff_size = ((int )(strlen(STONESOUP_TAINT_SOURCE)));
strncpy(stonesoup_heap_buff_64, STONESOUP_TAINT_SOURCE, 64);

for (; stonesoup_ss_i < stonesoup_buff_size; ++stonesoup_ss_i){
    /* STONESOUP: CROSSOVER-POINT (Buffer Overread) */
    /* STONESOUP: TRIGGER-POINT (Buffer Overread) */
    stonesoup_printf("%02x",stonesoup_heap_buff_64[stonesoup_ss_i]);
}
```

Benign Inputs

Any string of 64 characters or less is a benign input. For example:

```
AAAAAAAAAAAAAAAAAAAAAAA
```

```
Hello World!
```

```
Short is good
```

Due to the fact that none of these inputs are greater than the buffer size of 64 characters (63 char and one null byte) they will not result in a bad state.

Exploiting Inputs

Any string of 54 characters or greater is an exploiting input. For example:

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAA
```

```
This is probably going to overflow a buffer of some sort.  Let's run it anyway and
see what happens.  I hope that everything turns out ok, because I would love to see
what this looks like in all CAPS.  Converting ASCII characters to upper case is
easy, you can either call a system function, or just add the standard difference
between the start of the two character sets.  But, I am sure there is a reason that
system functions exist for this sort of functionality.
```

Since both of these strings are significantly greater than the buffer size of 64 characters (63 char and one null byte) they will overflow into random memory and cause an observable bad state.

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Memory Corruption
- Restrictions and Requirements

C - CWE-126C - Buffer Over-Read

Summary

This snippet implements an unchecked read from a buffer. The buffer is declared as a fixed size member of a struct that is allocated on the heap. Untrusted input is not properly sanitized or restricted before being used to determine the number of characters to read from the buffer. This allows input greater than 63 characters in length to over read the buffer, printing the contents of the memory located after the buffer on the heap.

For weakness C-CWE-126A, the buffer over-read occurs on the stack.

For weakness C-CWE-126B, the buffer over-read occurs on the heap.

For weakness C-CWE-126C, the buffer over-read occurs on the heap, printing an observable set within a user-defined struct.

For weakness C-CWE-126D, the buffer over-read occurs on the stack, printing an observable set within a user-defined struct.

These weaknesses differ from CWE120, because CWE120 is a buffer overwrite, and these are buffer over-reads.

These weaknesses differ from CWE127, because CWE127 is a buffer under-read, and these are buffer over-reads.

Weakness Class	Memory Corruption
CWE	CWE-126
Variant	A
Language	C
Status	WEAK-401 - CWE-126-2-C DONE
Negative Technical Impact(s)	Information Disclosure

Implementation

The snippet declares a character buffer of size 64 as part of a struct that gets allocated on the heap. The struct itself contains three character buffers of size 64, one before and one after the buffer used to implement the weakness, used as observables. The snippet also declares two integers, one to contain the size of the untrusted input string and the other set to 0 to be used as a loop counter later in execution.

```
int stonesoup_i = 0;
int stonesoup_buff_size = 0;

struct stonesoup_struct* stonesoup_data = NULL;

stonesoup_data = (struct stonesoup_struct*) malloc(sizeof(struct stonesoup_struct));
```

The snippet then fills the newly allocated buffer with 'Q's and fills the other two buffers with 'A's, null terminating all three. The snippet then gets the size of the input string and saves it to the variable declared above. It then safely copies the input string into the internal buffer and uses the size of the input string to iterate through the buffer, printing each character to the output stream. Since the size is not sanitized before being used to iterate over the buffer, an input string of 65 characters or more will cause the snippet to unintentionally disclose the data stored in memory addresses after the buffer on the heap, in this case a bunch of 'A's, possibly followed by more data if the over read is large enough. This is both the crossover and the trigger point.

```
memset(stonesoup_data->before, 'A', 63);
stonesoup_data->before[63] = '\0';

memset(stonesoup_data->buffer, 'Q', 63);
stonesoup_data->buffer[63] = '\0';

memset(stonesoup_data->after, 'A', 63);
stonesoup_data->after[63] = '\0';

stonesoup_buff_size = ((int )(strlen(STONESOUP_TAINT_SOURCE)));
memcpy(stonesoup_data->buffer, STONESOUP_TAINT_SOURCE, 64);

for (; stonesoup_i < stonesoup_buff_size; ++stonesoup_i){
    /* STONESOUP: CROSSOVER-POINT (Buffer Overread) */
    /* STONESOUP: TRIGGER-POINT (Buffer Overread) */
    stonesoup_printf("%x",stonesoup_data->buffer[stonesoup_i]);
}
```

Benign Inputs

Any string of 64 characters or less is a benign input. For example:

```
AAAAAAAAAAAAAAAAAAAAAAA
```

```
Hello World!
```

```
Short is good
```

Due to the fact that none of these inputs are greater than the buffer size of 64 characters (63 char and one null byte) they will not result in a bad state.

Exploiting Inputs

Any string of 54 characters or greater is an exploiting input. For example:

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAA
```

```
This is probably going to overflow a buffer of some sort.  Let's run it anyway and
see what happens.  I hope that everything turns out ok, because I would love to see
what this looks like in all CAPS.  Converting ASCII characters to upper case is
easy, you can either call a system function, or just add the standard difference
between the start of the two character sets.  But, I am sure there is a reason that
system functions exist for this sort of functionality.
```

Since both of these strings are significantly greater than the buffer size of 64 characters (63 char and one null byte) they will overflow into random memory and cause an observable bad state.

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Memory Corruption
- Restrictions and Requirements

C - CWE-126D - Buffer Over-Read

Summary

This snippet implements an unchecked read from a buffer. The buffer is declared as a fixed size member of a struct that is allocated on the stack. Untrusted input is not properly sanitized or restricted before being used to determine the number of characters to read from the buffer. This allows input greater than 63 characters in length to over read the buffer, printing the contents of the memory located after the buffer on the stack.

For weakness C-CWE-126A, the buffer over-read occurs on the stack.

For weakness C-CWE-126B, the buffer over-read occurs on the heap.

For weakness C-CWE-126C, the buffer over-read occurs on the heap, printing an observable set within a user-defined struct.

For weakness C-CWE-126D, the buffer over-read occurs on the stack, printing an observable set within a user-defined struct.

These weaknesses differ from CWE120, because CWE120 is a buffer overwrite, and these are buffer over-reads.

These weaknesses differ from CWE127, because CWE127 is a buffer under-read, and these are buffer over-reads.

Weakness Class	Memory Corruption
CWE	CWE-126
Variant	A
Language	C
Status	**WEAK-421** - CWE-126-3-C **DONE**
Negative Technical Impact(s)	Information Disclosure

Implementation

The snippet declares a character buffer of size 64 as part of a struct that gets allocated on the struct. The struct itself contains three character buffers of size 64, one before and one after the buffer used to implement the weakness, used as observables. The snippet also declares two integers, one to contain the size of the untrusted input string and the other set to 0 to be used as a loop counter later in execution.

```
int stonesoup_i = 0;
int stonesoup_buff_size = 0;

struct stonesoup_struct stonesoup_data;
```

The snippet then fills the newly allocated buffer with 'Q's and fills the other two buffers with 'A's, null terminating all three. The snippet then gets the size of the input string and saves it to the variable declared above. It then safely copies the input string into the internal buffer and uses the size of the input string to iterate through the buffer, printing each character to the output stream. Since the size is not sanitized before being used to iterate over the buffer, an input string of 65 characters or more will cause the snippet to unintentionally disclose the data stored in memory addresses after the buffer on the stack, in this case a bunch of 'A's, possibly followed by more data if the over read is large enough. This is both the crossover and the trigger point.

```
for (stonesoup_i = 0; stonesoup_i < 63; stonesoup_i++) {
    stonesoup_data.before[stonesoup_i] = 'A';
}
stonesoup_data.before[63] = '\0';

for (stonesoup_i = 0; stonesoup_i < 63; stonesoup_i++) {
    stonesoup_data.buffer[stonesoup_i] = 'Q';
}
stonesoup_data.buffer[63] = '\0';

for (stonesoup_i = 0; stonesoup_i < 63; stonesoup_i++) {
    stonesoup_data.after[stonesoup_i] = 'A';
}
stonesoup_data.after[63] = '\0';


stonesoup_buff_size = ((int )(strlen(STONESOUP_TAINT_SOURCE)));
memcpy(stonesoup_data.buffer, STONESOUP_TAINT_SOURCE, 64);

for (; stonesoup_i < stonesoup_buff_size; ++stonesoup_i){
    /* STONESOUP: CROSSOVER-POINT (Buffer Overread) */
    /* STONESOUP: TRIGGER-POINT (Buffer Overread) */
    stonesoup_printf("%x",stonesoup_data.buffer[stonesoup_i]);
}
```

Benign Inputs

Any string of 64 characters or less is a benign input. For example:

```
AAAAAAAAAAAAAAAAAAAAAAA
```

```
Hello World!
```

```
Short is good
```

Due to the fact that none of these inputs are greater than the buffer size of 64 characters (63 char and one null byte) they will not result in a bad state.

Exploiting Inputs

Any string of 54 characters or greater is an exploiting input. For example:

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAA
```

```
This is probably going to overflow a buffer of some sort.  Let's run it anyway and
see what happens.  I hope that everything turns out ok, because I would love to see
what this looks like in all CAPS.  Converting ASCII characters to upper case is
easy, you can either call a system function, or just add the standard difference
between the start of the two character sets.  But, I am sure there is a reason that
system functions exist for this sort of functionality.
```

Since both of these strings are significantly greater than the buffer size of 64 characters (63 char and one null byte) they will overflow into random memory and cause an observable bad state.

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Memory Corruption
- Restrictions and Requirements

C - CWE-127A - Buffer Under-Read

Summary

This snippet implements an unchecked read from a buffer. The buffer is declared as a fixed size buffer on the stack. Untrusted input is not properly sanitized or restricted before being used as an index of the buffer to read. This allows inputs containing non-ascii characters to wrap around to negative indexes and under read the buffer, printing the contents of the memory located before the buffer on the stack to the output stream.

Weakness Class	Memory Corruption
CWE	CWE-127
Variant	A

Language	C
Status	WEAK-132 - CWE-127-0-C DONE
Negative Technical Impact(s)	Information Disclosure

Implementation

The snippet allocates an integer buffer of size 128 on the stack initialized to all '0's and declares a second integer buffer of size 200 before the first on the stack, initialized to all '5555's. The first is used to contain the number of times each ASCII character appears in the input, mapping each ASCII value to it's equivalent index. The second buffer is set to contain '5555' at each index in order to provide an observable in the event that the weakness occurs and the first buffer under reads.

```
int stonesoup_stack_buff[128];
int stonesoup_other_buff[200];
```

The snippet takes two strings as input, one via the source taint and one via an environment variable 'INPUT_STRING'. Once executing, the snippet iterates though the string passed via the 'INPUT_STRING' environment variable and increments the corresponding index in the first buffer for each character in the input string.

```
for (stonesoup_ss_i = 0; stonesoup_ss_i < strlen((char *)stonesoup_input_string);
++stonesoup_ss_i) {
 if (stonesoup_input_string[stonesoup_ss_i] < 0)
  continue;
 ++stonesoup_stack_buff[stonesoup_input_string[stonesoup_ss_i]];
}
```

The snippet then takes the string passed via the source taint and iterates though each character, using that character as the index to the first buffer as above. The snippet then prints the data contained in that buffer. If the character being used as an index is an extended ASCII character, the value will overflow into negative and underread the buffer, printing the sensitive data to the output stream. This is both the crossover and the trigger point.

```
for (stonesoup_ss_i = 0; stonesoup_ss_i < strlen(STONESOUP_TAINT_SOURCE);
++stonesoup_ss_i) {
 /* STONESOUP: CROSSOVER-POINT (Buffer Underread) */
 /* STONESOUP: TRIGGER-POINT (Buffer Underread) */
 stonesoup_printf("value %c appears: %d times\n",
     STONESOUP_TAINT_SOURCE[stonesoup_ss_i],
     stonesoup_stack_buff[(int) STONESOUP_TAINT_SOURCE[stonesoup_ss_i]]);
}
```

If the input string does not contain extended ASCII characters a buffer under read will not happen and no information will be disclosed.

Benign Inputs

Any two input strings not containing extended ASCII characters. For example:

```
Source Taint:
AAAABVVCUYEOAISDHGUEHGU:WOJOQIUFGUYGBJdjsafihuegakdlfm9238946665232kmemiucyds78fy2

INPUT_STRING:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

```
Source Taint:
AAAABVVCUYEOAISDHGUEHGU:WOJOQIUFGUYGBJdjsafihuegakdlfm9238946665232kmemiucyds78fy2

INPUT_STRING: This is probably not going to overflow a buffer of some sort.  Let's
run it anyway and see what happens.
```

```
Source Taint:
AAAABVVCUYEOAISDHGUEHGU:WOJOQIUFGUYGBJdjsafihuegakdlfm9238946665232kmemiucyds78fy2

INPUT_STRING: As long as this statement has more than 64 characters, then nothing
bad should happen
```

Since none of these inputs contain extended ASCII characters they will not result in a bad state.

Exploiting Inputs

Any string in the 'INPUT_STRING' variable not containing extended ASCII characters with an input string (source taint) containing extended ASCII characters. For example:

```
Source Taint: QUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFBQUGugIE=
Where this is a base64 encoded string.

INPUT_STRING:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

```
Source Taint: SGVsbKUgV6lybGSI
Where this is a base64 encoded string.

INPUT_STRING:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

Since both of these strings contained in the source taint have extended ASCII characters they will cause a buffer under read.

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Memory Corruption
- Restrictions and Requirements

C - CWE-127B - Buffer Under-Read

Summary

This snippet implements an incorrect size check when reading from a buffer that can cause a buffer under read. The buffer is declared as a character buffer of size 64 on the heap. 64 bytes of input are copied into the buffer but the snippet incorrectly uses the original size of the input string to read the buffer to the output stream in reverse order leading to a buffer under read.

Weakness Class	Memory Corruption
CWE	CWE-127
Variant	B
Language	C
Status	WEAK-134 - CWE-127-1-C DONE
Negative Technical Impact(s)	Information Disclosure

Implementation

The snippet allocates an character buffer of size 64 on the heap and stores the size of the input string and the size of the allocated buffer.

```
char * stonesoup_other_buff = 0;
int stonesoup_buff_size = 0;
int stonesoup_other_size = 0;
int stonesoup_size;
stonesoup_buff_size = strlen(STONESOUP_TAINT_SOURCE) + 1;

stonesoup_other_size = 64;
stonesoup_other_buff = (char*) malloc (stonesoup_other_size * sizeof (char));
```

Once executing, the snippet saves the size of either the allocated buffer or the size of the input string, whichever is smaller, and copies this number of characters from the input string into the buffer in reverse order.

```
stonesoup_size = stonesoup_other_size < stonesoup_buff_size ? stonesoup_other_size :
stonesoup_buff_size;


for (stonesoup_i = 0; stonesoup_i < stonesoup_size; stonesoup_i++) {
    stonesoup_other_buff[stonesoup_other_size - stonesoup_i - 1] =
        STONESOUP_TAINT_SOURCE[stonesoup_buff_size - stonesoup_i - 1];
}
```

The snippet then reads the characters from the internal buffer, also from highest index to lowest, however it incorrectly uses the size of the input string instead of the size of the string contained in the buffer. This is both the crossover and trigger point and causes any input string of 64 characters or more to result in a buffer under read.

```
for (stonesoup_i = 0; stonesoup_i < stonesoup_buff_size; stonesoup_i++) {
        /* STONESOUP: CROSSOVER-POINT (Buffer Underread) */
        /* STONESOUP: TRIGGER-POINT (Buffer Underread) */
        stonesoup_printf("%02x",stonesoup_other_buff[stonesoup_other_size -
stonesoup_i - 1]);
}
```

If the input string is 63 characters or less in length then a buffer under read will not occur and the snippet will complete without error.

Benign Inputs

Any input string with 63 or less characters is benign. For example:

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

```
This is probably not going to overflow a buffer of some sort.
```

```
This statement has less than 64 characters, then nothing bad.
```

Since none of these inputs contain 64 or more characters they will not result in a bad state.

Exploiting Inputs

Any input string containing 64 or more characters is exploiting. For example:

```
abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdef
ghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz
```

```
The quick brown fox jumps over the lazy dog.  Why?  I don't know, because he was in
a rush and trying to get somewhere.
```

Since both of these strings contained in the source taint have extended ASCII characters they will cause a buffer under read.

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Memory Corruption
- Restrictions and Requirements

C - CWE-127C - Buffer Under-Read

Summary

This snippet implements an unchecked read from a buffer. The buffer is declared as a fixed size buffer as part of a struct on the heap. Untrusted input is not properly sanitized or restricted before being used as an index of the buffer to read. This allows inputs containing non-ascii characters to wrap around to negative indexes and under read the buffer, printing the contents of the memory located before the buffer on the heap to the output stream.

Weakness Class	Memory Corruption
CWE	CWE-127
Variant	C
Language	C
Status	WEAK-402 - CWE-127-2-C DONE
Negative Technical Impact(s)	Information Disclosure

Implementation

The snippet uses an integer buffer of size 128 located in a struct allocated on the heap, initialized to all '0's, and a second integer buffer of size 200 before the first located in the same struct allocated on the heap, initialized to all '5555's. The first is used to contain the number of times each ASCII character appears in the input, mapping each ASCII value to it's equivalent index. The second buffer is set to contain '5555' at each index in order to provide an observable in the event that the weakness occurs and the first buffer under reads.

```
struct stonesoup_struct {
    int before[200];
    int buffer[128];
    int after[200];
};

{...}

    stonesoup_data = (struct stonesoup_struct *) malloc (sizeof (struct
stonesoup_struct));
    if (stonesoup_data != NULL) {
        if (stonesoup_input_string != 0) {
            memset(stonesoup_data->buffer, 0, 128);
            for (stonesoup_i = 0; stonesoup_i < 200; ++stonesoup_i) {
                stonesoup_data->before[stonesoup_i] = 5555;
                stonesoup_data->after[stonesoup_i] = 5555;
            }
```

The snippet takes two strings as input, one via the source taint and one via an environment variable 'INPUT_STRING'. Once executing, the snippet iterates though the string passed via the 'INPUT_STRING' environment variable and increments the corresponding index in the first buffer for each character in the input string.

```
for (stonesoup_i = 0; stonesoup_i < strlen((char *) stonesoup_input_string);
++stonesoup_i) {
    if (stonesoup_input_string[stonesoup_i] < 0)
        continue;
     ++stonesoup_data->buffer[stonesoup_input_string[stonesoup_i]];
}
```

The snippet then takes the string passed via the source taint and iterates though each character, using that character as the index to the first buffer as above. The snippet then prints the data contained in that buffer. If the character being used as an index is an extended ASCII character, the value will overflow into negative and under read the buffer, printing the sensitive data to the output stream. This is both the crossover and the trigger point.

```
for (stonesoup_i = 0; stonesoup_i < strlen(STONESOUP_TAINT_SOURCE); ++stonesoup_i) {
    /* STONESOUP: CROSSOVER-POINT (Buffer Underread) */
    /* STONESOUP: TRIGGER-POINT (Buffer Underread) */
    stonesoup_printf("value %c appears: %d times\n",
        STONESOUP_TAINT_SOURCE[stonesoup_i],
        stonesoup_data->buffer[(int) STONESOUP_TAINT_SOURCE[stonesoup_i]]);
}
```

If the input string does not contain extended ASCII characters a buffer under read will not happen and no information will be disclosed.

Benign Inputs

Any two input strings not containing extended ASCII characters. For example:

```
Source Taint:
AAAABVVCUYEOAISDHGUEHGU:WOJOQIUFGUYGBJdjsafihuegakdlfm9238946665232kmemiucyds78fy2

INPUT_STRING:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

```
Source Taint:
AAAABVVCUYEOAISDHGUEHGU:WOJOQIUFGUYGBJdjsafihuegakdlfm9238946665232kmemiucyds78fy2

INPUT_STRING: This is probably not going to overflow a buffer of some sort.  Let's
run it anyway and see what happens.
```

```
Source Taint:
AAAABVVCUYEOAISDHGUEHGU:WOJOQIUFGUYGBJdjsafihuegakdlfm9238946665232kmemiucyds78fy2

INPUT_STRING: As long as this statement has more than 64 characters, then nothing
bad should happen
```

Since none of these inputs contain extended ASCII characters they will not result in a bad state.

Exploiting Inputs

Any string in the 'INPUT_STRING' variable not containing extended ASCII characters with an input string (source taint) containing extended ASCII characters. For example:

```
Source Taint: QUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFBQUGugIE=
Where this is a base64 encoded string.

INPUT_STRING:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

```
Source Taint: SGVsbKUgV6lybGSI
Where this is a base64 encoded string.

INPUT_STRING:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

Since both of these strings contained in the source taint have extended ASCII characters they will cause a buffer under read.

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Memory Corruption
- Restrictions and Requirements

C - CWE-127D - Buffer Under-Read

Summary

This snippet implements an unchecked read from a buffer. The buffer is declared as a fixed size buffer as part of a struct on the stack. Untrusted input is not properly sanitized or restricted before being used as an index of the buffer to read. This allows inputs containing non-ascii characters to wrap around to negative indexes and under read the buffer, printing the contents of the memory located before the buffer on the stack to the output stream.

Weakness Class	Memory Corruption
CWE	CWE-127
Variant	D
Language	C
Status	**WEAK-422** - CWE-127-3-C DONE
Negative Technical Impact(s)	Information Disclosure

Implementation

The snippet uses an integer buffer of size 128 located in a struct allocated on the stack, initialized to all '0's, and a second integer buffer of size 200 before the first located in the same struct allocated on the stack, initialized to all '5555's. The first is used to contain the number of times each ASCII character appears in the input, mapping each ASCII value to it's equivalent index. The second buffer is set to contain '5555' at each index in order to provide an observable in the event that the weakness occurs and the first buffer under reads.

```
struct stonesoup_struct stonesoup_data;
int stonesoup_i = 0;
stonesoup_input_string = (signed char *) getenv(STONESOUP_EXPRESSION_1);
if (stonesoup_input_string != 0) {
    for (stonesoup_i = 0; stonesoup_i < 128; ++stonesoup_i) {
        stonesoup_data.buffer[stonesoup_i] = 0;
    }
    for (stonesoup_i = 0; stonesoup_i < 200; ++stonesoup_i) {
        stonesoup_data.before[stonesoup_i] = 5555;
        stonesoup_data.after[stonesoup_i] = 5555;
    }
```

The snippet takes two strings as input, one via the source taint and one via an environment variable 'INPUT_STRING'. Once executing, the snippet iterates though the string passed via the 'INPUT_STRING' environment variable and increments the corresponding index in the first buffer for each character in the input string.

```
for (stonesoup_i = 0; stonesoup_i < strlen((char *) stonesoup_input_string);
++stonesoup_i) {
    if (stonesoup_input_string[stonesoup_i] < 0)
        continue;
    ++stonesoup_data.buffer[stonesoup_input_string[stonesoup_i]];
}
```

The snippet then takes the string passed via the source taint and iterates though each character, using that character as the index to the first buffer as above. The snippet then prints the data contained in that buffer. If the character being used as an index is an extended ASCII character, the value will overflow into negative and under read the buffer, printing the sensitive data to the output stream. This is both the crossover and the trigger point.

```
for (stonesoup_i = 0; stonesoup_i < strlen(STONESOUP_TAINT_SOURCE); ++stonesoup_i) {
    /* STONESOUP: CROSSOVER-POINT (Buffer Underread) */
    /* STONESOUP: TRIGGER-POINT (Buffer Underread) */
    stonesoup_printf("value %c appears: %d times\n",
        STONESOUP_TAINT_SOURCE[stonesoup_i],
        stonesoup_data.buffer[(int) STONESOUP_TAINT_SOURCE[stonesoup_i]]);
}
```

If the input string does not contain extended ASCII characters a buffer under read will not happen and no information will be disclosed.

Benign Inputs

Any two input strings not containing extended ASCII characters. For example:

```
Source Taint:
AAAABVVCUYEOAISDHGUEHGU:WOJOQIUFGUYGBJdjsafihuegakdlfm9238946665232kmemiucyds78fy2

INPUT_STRING:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

```
Source Taint:
AAAABVVCUYEOAISDHGUEHGU:WOJOQIUFGUYGBJdjsafihuegakdlfm9238946665232kmemiucyds78fy2

INPUT_STRING: This is probably not going to overflow a buffer of some sort.  Let's
run it anyway and see what happens.
```

```
Source Taint:
AAAABVVCUYEOAISDHGUEHGU:WOJOQIUFGUYGBJdjsafihuegakdlfm9238946665232kmemiucyds78fy2

INPUT_STRING: As long as this statement has more than 64 characters, then nothing
bad should happen
```

Since none of these inputs contain extended ASCII characters they will not result in a bad state.

Exploiting Inputs

Any string in the 'INPUT_STRING' variable not containing extended ASCII characters with an input string (source taint) containing extended ASCII characters. For example:

```
Source Taint: QUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFBQUGugIE=
Where this is a base64 encoded string.

INPUT_STRING:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

```
Source Taint: SGVsbKUgV6lybGSI
Where this is a base64 encoded string.

INPUT_STRING:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

Since both of these strings contained in the source taint have extended ASCII characters they will cause a buffer under read.

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Memory Corruption
- Restrictions and Requirements

C - CWE-129A - Improper Validation of Array Index

Summary

This snippet implements an improper array index validation that can cause a function pointer to get overwritten leading to a segfault. The snippet takes untrusted user input and uses it to calculate array indexes which then get modified. If the untrusted input contains certain ASCII characters the array index calculation will result in a negative array index, modifying memory that it should not be accessing.

Weakness Class	Memory Corruption
CWE	CWE-129
Variant	A
Language	C
Status	WEAK-143 - CWE-129-0-C DONE
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

The snippet allocates a character array of size 62 on the stack and creates two function pointers, one before and one after the character array on the stack. These two function pointers are set to either `stonesoup_canary_function_1()` or `stonesoup_canary_function_2()` based on the input.

```
int (*stonesoup_after_ptr[1])();
unsigned char stonesoup_count[62];
int (*stonesoup_before_ptr[1])();

{...}

if (strlen(stonesoup_str_buf) > 1 && stonesoup_str_buf[0] > 'a') {
    stonesoup_before_ptr[0] = stonesoup_canary_function_1;
    stonesoup_after_ptr[0] = stonesoup_canary_function_1;
}
else {
    stonesoup_before_ptr[0] = stonesoup_canary_function_2;
    stonesoup_after_ptr[0] = stonesoup_canary_function_2;
}
```

Once executing, the snippet iterates through the input string, and for each character calculates a string index and uses this index to increment the character array. This index is calculated such that any ASCII character with a value of less than '48' will result in a negative array index, which can corrupt memory before the array, possibly leading to a segfault. This is both the crossover and the trigger point.

```
for (stonesoup_i = 0; stonesoup_i < strlen(stonesoup_str_buf); stonesoup_i++)
    /* STONESOUP: CROSSOVER-POINT (Improper Validation of Array Index) */
    /* STONESOUP: TRIGGER-POINT (Improper Validation of Array Index: Ascii Bounds)
*/
{
    if (stonesoup_str_buf[stonesoup_i] > 96) {
        stonesoup_index = stonesoup_str_buf[stonesoup_i] - 'a' + 36;
        if (stonesoup_count[stonesoup_index] < 255)
            stonesoup_count[stonesoup_index]++;
    }
    else if (stonesoup_str_buf[stonesoup_i] < 58) {
        stonesoup_index = stonesoup_str_buf[stonesoup_i] - 48;
        if (stonesoup_count[stonesoup_index] < 255)
            stonesoup_count[stonesoup_index]++;
    }
    else {
        stonesoup_index = stonesoup_str_buf[stonesoup_i] - 'A' + 10;
        if (stonesoup_count[stonesoup_index] < 255)
            stonesoup_count[stonesoup_index]++;
    }
}
```

Once the snippet has finished iterating through the input string, it iterates though the character array printing the value at each index. The snippet then calls the two function pointers, and if a negative array index was calculated earlier such that the stonesoup_before_ptr[0] was corrupted the snippet will result in a segfault.

```
for (stonesoup_i = 0; stonesoup_i < 62; stonesoup_i++) {
    stonesoup_printf("index %d: %d\n",stonesoup_i,stonesoup_count[stonesoup_i]);
}
stonesoup_printf("%d\n",stonesoup_before_ptr[0]());
stonesoup_printf("%d\n",stonesoup_after_ptr[0]());
```

If the pointer was not corrupted, then the snippet will finish execution without error.

Benign Inputs

Any input string that does not contain ASCII characters with values of less than 48 is benign. For example:

```
exampleofgoodinput
```

```
abcd12345ABCDE
```

```
R0G3RR4881T
```

Since none of these inputs contain ASCII characters with values less than 48 they will not result in a bad state.

Exploiting Inputs

Any string that contains ASCII characters with values less than 48 such that, for a character C, C - 48 is some negative index that will cause the snippet to corrupt the `stonesoup_before_ptr[0]`. For example:

```
{a+b|c+d}
```

```
!a*b(c*d-1)
```

Since both of these strings contain ASCII characters that will cause `stonesoup_before_ptr[0]` to become corrupt, they will both cause a segfault.

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Memory Corruption
- Restrictions and Requirements

C - CWE-129B - Improper Validation of Array Index

Summary

This snippet implements an improper array index validation that can cause a function pointer to get overwritten leading to a segfault. The snippet takes untrusted user input and uses it to calculate array indexes which then get modified. If the untrusted input contains certain ASCII characters the array index calculation will result in a negative array index, modifying memory that it should not be accessing.

Weakness Class	Memory Corruption
CWE	CWE-129
Variant	B
Language	C
Status	**WEAK-146** - CWE-129-1-C **DONE**
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

The snippet allocates a character array of size 62 on the heap and creates two char**s, one before and one after the character array on the stack. These two char**s both point to char*s allocated on the heap which in turn both point to the input string.

```
char **stonesoup_ptr_after = 0;
unsigned char *stonesoup_count = 0;
char **stonesoup_ptr_before = 0;

{...}

stonesoup_ptr_after = malloc(sizeof(char *));
if (stonesoup_ptr_after == 0) {
    stonesoup_printf("Error: Failed to allocate memory\n");
    exit(1);
}
stonesoup_count = malloc(sizeof(unsigned char ) * 62);
if (stonesoup_count == 0) {
    stonesoup_printf("Error: Failed to allocate memory\n");
    exit(1);
}
stonesoup_ptr_before = malloc(sizeof(char *));
if (stonesoup_ptr_before == 0) {
    stonesoup_printf("Error: Failed to allocate memory\n");
    exit(1);
}
*stonesoup_ptr_before = STONESOUP_TAINT_SOURCE;
*stonesoup_ptr_after = STONESOUP_TAINT_SOURCE;
```

Once executing, the snippet iterates through the input string, and for each character calculates a string index and uses this index to increment the character array. This index is calculated such that any ASCII character with a value of less than '48' will result in a negative array index, which can corrupt memory before the array, possibly leading to a segfault. This is both the crossover and the trigger point.

```
for (stonesoup_i = 0; stonesoup_i < strlen(STONESOUP_TAINT_SOURCE); stonesoup_i++) {
    /* STONESOUP: CROSSOVER-POINT (Improper Validation of Array Index) */
    /* STONESOUP: TRIGGER-POINT (Improper Validation of Array Index: Ascii Bounds)
*/
    if (STONESOUP_TAINT_SOURCE[stonesoup_i] > 96) {
        stonesoup_index = STONESOUP_TAINT_SOURCE[stonesoup_i] - 'a' + 36;
        if (stonesoup_count[stonesoup_index] < 255)
            stonesoup_count[stonesoup_index]++;
    }
    else if (STONESOUP_TAINT_SOURCE[stonesoup_i] < 58) {
        stonesoup_index = STONESOUP_TAINT_SOURCE[stonesoup_i] - 48;
        if (stonesoup_count[stonesoup_index] < 255)
            stonesoup_count[stonesoup_index]++;
    }
    else {
        stonesoup_index = STONESOUP_TAINT_SOURCE[stonesoup_i] - 'A' + 10;
        if (stonesoup_count[stonesoup_index] < 255)
            stonesoup_count[stonesoup_index]++;
    }
}
```

Once the snippet has finished iterating through the input string, it iterates though the character array printing the value at each index. The snippet then attempts to print the length of the string pointed to by `stonesoup_ptr_before`. If this pointer was corrupted by the above trigger point, this call will result in a segfault.

```
for (stonesoup_i = 0; stonesoup_i < 62; stonesoup_i++)
    stonesoup_printf("index %d: %d\n",stonesoup_i,stonesoup_count[stonesoup_i]);
stonesoup_printf("%d %d\n",strlen( *stonesoup_ptr_before),strlen(
*stonesoup_ptr_after));
```

If the pointer was not corrupted, then the snippet will finish execution without error.

Benign Inputs

Any input string that does not contain ASCII characters with values of less than 48 is benign. For example:

```
exampleofgoodinput
```

```
abcd12345ABCDE
```

```
R0G3RR4881T
```

Since none of these inputs contain ASCII characters with values less than 48 they will not result in a bad state.

Exploiting Inputs

Any string that contains ASCII characters with values less than 48 such that, for a character C, C - 48 is some negative index that will cause the snippet to corrupt the `stonesoup_ptr_before`. For example:

```
a-b+c*d
```

```
3.14159
```

Since both of these strings contain ASCII characters that will cause `stonesoup_ptr_before` to become corrupt, they will both cause a segfault.

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Memory Corruption
- Restrictions and Requirements

C - CWE-134A - Uncontrolled Format String

Summary

This snippet implements an sprintf that uses untrusted user input without a format string. The snippet takes untrusted user input and passes it

to an sprintf that does not implement a format string. This allows the user to pass format strings to the snippet causing it to leak sensitive data and possibly allowing an attacker to cause a buffer overflow and inject shell code.

Weakness Class	Memory Corruption
CWE	CWE-134
Variant	A
Language	C
Status	WEAK-168 - CWE-134-0-C DONE
Negative Technical Impact(s)	Information Exposure

Implementation

The snippet allocates a character array of size 128 on the stack and makes a call to sprintf to copy the input string into the character array. This sprintf call uses the users input in place of a format string allowing the user to pass a format string as input. This is both the crossover and the trigger point.

```
char stonesoup_buffer_stack[128] = {0};
/* STONESOUP: CROSSOVER-POINT (Uncontrolled Format String) */
/* STONESOUP: TRIGGER-POINT (Uncontrolled Format String) */
sprintf(stonesoup_buffer_stack,STONESOUP_TAINT_SOURCE);
stonesoup_printf("%s\n",stonesoup_buffer_stack);
```

If the input string is a specially formed format string, format string expansion will be preformed on it by sprintf, and the result of this will be printed to the output stream by the call to `stonesoup_printf()` at the end of the snippets execution. This can lead to information exposure.

Benign Inputs

Any input string that does not contain formatting. For example:

```
this is a string to be printed
```

```
c29tZSBiZW5pZ24gZm9ybWF0dGluZyBmb3IgY2hhciBzdHJpbmcgCg

Where the above text is base64 encoded.
```

```
Cg==

Where the above text is base64 encoded.
```

Since none of these inputs contain formatting they will not result in a bad state.

Exploiting Inputs

Any string that containing formatting that will cause information exposure. For example:

```
%0500d
```

```
%.08x%.08x%.08x%.08x%.08x%.08x%.08x%.08x%.08x%.08x%.08x%.08x%.08x%.08x%.08x
```

Since both of these strings contain formatting that will lead to information exposure, they are exploiting input.

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Memory Corruption
- Restrictions and Requirements

C - CWE-134B - Uncontrolled Format String

Summary

This snippet implements an sprintf that uses untrusted user input without a format string. The snippet takes untrusted user input and passes it to an sprintf that does not implement a format string. This allows the user to pass format strings to the snippet causing it to leak sensitive data and possibly allowing an attacker to cause a buffer overflow and inject shell code.

Weakness Class	Memory Corruption
CWE	CWE-134
Variant	B
Language	C
Status	WEAK-170 - CWE-134-1-C DONE
Negative Technical Impact(s)	Information Exposure

Implementation

The snippet allocates a character array of size 128 on the heap, initialized to all '0's, and and creates a char** to point to this buffer.

```
char *stonesoup_buffer_heap = 0;
char **stonesoup_buf_ptr = 0;
stonesoup_buffer_heap = malloc(sizeof(char ) * 128);
if (stonesoup_buffer_heap == 0) {
    stonesoup_printf("Error: Failed to allocate memory\n");
    exit(1);
}
memset(stonesoup_buffer_heap,0,128);
stonesoup_buf_ptr = malloc(sizeof(char *));
if (stonesoup_buf_ptr == 0) {
    stonesoup_printf("Error: Failed to allocate memory\n");
    exit(1);
}
*stonesoup_buf_ptr = stonesoup_buffer_heap;
```

The snippet then calls sprintf to copy the input string into the character array on the heap, but incorrectly uses the input string in place of a format string. This allows a user to supply a custom format string leading to possible information exposure. The input string gets evaluated as a format string, and it's result is then printed to the output stream. This is both the crossover and the trigger point.

```
/* STONESOUP: CROSSOVER-POINT (Uncontrolled Format String) */
/* STONESOUP: TRIGGER-POINT (Uncontrolled Format String) */
sprintf(stonesoup_buffer_heap,STONESOUP_TAINT_SOURCE);
stonesoup_printf("%s\n",stonesoup_buffer_heap);
```

The character array is the converted to upper case and printed to the output stream again.

```
for (; stonesoup_oc_i < stonesoup_len; ++stonesoup_oc_i) {
    stonesoup_buffer_heap[stonesoup_oc_i] =
stonesoup_toupper(stonesoup_buffer_heap[stonesoup_oc_i]);
}
stonesoup_printf("%s\n",stonesoup_buffer_heap);
```

The snippet then finishes it's execution by `free()`ing the previously allocated array.

Benign Inputs

Any input string that does not contain formatting. For example:

```
this is a string to be printed
```

```
c29tZSBiZW5pZ24gZm9ybWF0dGluZyBmb3IgY2hhciBzdHJpbmcgCg

Where the above text is base64 encoded.
```

```
Cg==

Where the above text is base64 encoded.
```

Since none of these inputs contain formatting they will not result in a bad state.

Exploiting Inputs

Any string that containing formatting that will cause information exposure. For example:

```
%0500d
```

```
%175d
```

Since both of these strings contain formatting that will lead to information exposure, they are exploiting input.

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Memory Corruption
- Restrictions and Requirements

C - CWE-170A - Improper Null Termination

Summary

This snippet implements an file read of 128 characters which does not properly null terminate the copied string if the original string is 128 characters in length or greater. The snippet takes the name of a file, opens the file, and reads up to the first 128 characters into an internal buffer of size 128, allocated on the stack. This buffer is then copied via strcpy into a second stack allocated buffer of size 128, however, if the original file read did not copy a null terminator, the strcpy will copy everything it finds in memory after the input string until it encounters a null terminator. This will cause a buffer overflow and lead to a segfault.

Weakness Class	Memory Corruption
CWE	CWE-170
Variant	A
Language	C
Status	**WEAK-173** - CWE-170-0-C **DONE**
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

The snippet allocates two character arrays of size 128 on the stack. It then takes a file name as input and opens this file for reading.

```
char stonesoup_buffer[128];
char stonesoup_input_buf[128] = {0};
memset(stonesoup_buffer,'x',128);
stonesoup_buffer[127] = 0;
stonesoup_file_desc = open(STONESOUP_TAINT_SOURCE,0);
```

The snippet then reads the data from the file, up to 128 characters, and stores them in the second character array. If the file contains 128 characters or more, this will not copy over a null terminator to the end of the character array. This is the crossover point.

```
/* STONESOUP: CROSSOVER-POINT (Improper Null Termination) */
read(stonesoup_file_desc,stonesoup_input_buf,128);
```

The snippet then proceeds to copy this text into the first character array by calling strcpy. If the second array does not contain a null terminator, strcpy will copy the string, and all the memory after it up to the first null byte, and put this into the first array overflowing the buffer and overwriting random memory. This is the trigger point and can result in a segfault.

```
/* STONESOUP: TRIGGER-POINT (Improper Null Termination: Stack Over Read) */
strcpy(stonesoup_buffer,stonesoup_input_buf);
```

If the input file contained 127 characters or less a null terminator would have been read, and the buffer overflow that possibly lead to a segfault would not have occurred.

Benign Inputs

The name of any file containing a string of 127 characters or less or the name of any file that does not exist is benign. For example:

```
/opt/stonesoup/workspace/testData/good01.txt

Where the above is a file containing the text "This file is not empty."
```

```
/opt/stonesoup/workspace/testData/good02.txt


Where the above is a file containing the text
"abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcde
fghijklmnopqrstuvwxyzabcdefghijklmnopqrstuv"
```

```
/opt/stonesoup/workspace/testData/badfilename.txt


Where the above is a file that does not exist.
```

Exploiting Inputs

The name of any file containing 128 characters or more is exploiting. For example:

```
/opt/stonesoup/workspace/testData/bad01.txt


Where the above is a file containing the text
"aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaa"
```

```
/opt/stonesoup/workspace/testData/bad02.txt


Where the above is a file containing the text
"abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcde
fghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwx"
```

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Memory Corruption
- Restrictions and Requirements

C - CWE-170B - Improper Null Termination

Summary

This snippet implements an file read of 16 characters which does not properly null terminate the copied string if the original string is 16 characters in length or greater. The snippet takes the name of a file, opens the file, and reads up to the first 16 characters into an internal buffer of size 16, allocated on the stack. This buffer is then copied via strcpy into a heap allocated buffer of size 64, however, if the original file read did not copy a null terminator, the strcpy will copy everything it finds in memory after the input string until it encounters a null terminator. This will cause a buffer overflow and lead to a segfault.

Weakness Class	Memory Corruption
CWE	CWE-170
Variant	B

Language	C
Status	WEAK-174 - CWE-170-1-C DONE
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

The snippet allocates three character arrays, one static array containing the text "new test input", one of size 16 allocated on the stack, and one of size 64 allocated on the heap. The snippet also allocates a heap pointer that is set to point to the heap allocated pointer. It then takes a file name as input and opens this file for reading.

```
char *stonesoup_new_input = "new test input";
const int stonesoup_MAXLEN = 16;
int stonesoup_file_desc;
char stonesoup_input_buf[stonesoup_MAXLEN];
char **stonesoup_buf_ptr = 0;
char *stonesoup_path_buf = 0;
stonesoup_path_buf = malloc(sizeof(char ) * 64);
{...}
stonesoup_buf_ptr = malloc(sizeof(char *));
{...}
*stonesoup_buf_ptr = stonesoup_path_buf;
{...}
stonesoup_file_desc = open(STONESOUP_TAINT_SOURCE,0);
```

The snippet then reads the data from the file, up to 16 characters, and stores them in the second character array. If the file contains 16 characters or more, this will not copy over a null terminator to the end of the character array. The snippet the copies this safely into the heap array, but again does not append a null terminator to the end of the string. This is the crossover point.

```
read(stonesoup_file_desc,stonesoup_input_buf,stonesoup_MAXLEN);
close(stonesoup_file_desc);
/* STONESOUP: CROSSOVER-POINT (Improper Null Termination) */
strncpy(stonesoup_path_buf,stonesoup_input_buf,stonesoup_MAXLEN);
```

The snippet then proceeds concatenate the static string created above to the end of the heap character array. If the heap array does not contain a null terminator, the strcat will search forward through memory until it finds a null terminator and append the string to the memory location where it finds this, causing random memory to be overwritten and leading to a segfault. This is the trigger point.

```
/* STONESOUP: TRIGGER-POINT (Improper Null Termination: Heap Over Write) */
strcat(stonesoup_path_buf,stonesoup_new_input);
```

If the input file contained 15 characters or less a null terminator would have been read, and the buffer overflow would not have occurred.

Benign Inputs

The name of any file containing a string of 15 characters or less or the name of any file that does not exist is benign. For example:

```
/opt/stonesoup/workspace/testData/good01.txt

Where the above is a file containing the text "Hello World"
```

```
/opt/stonesoup/workspace/testData/good02.txt


Where the above is a file containing the text "This"
```

```
/opt/stonesoup/workspace/testData/badfilename.txt


Where the above is a file that does not exist.
```

Exploiting Inputs

The name of any file containing 128 characters or more is exploiting. For example:

```
/opt/stonesoup/workspace/testData/bad01.txt


Where the above is a file containing the text "abcdefghijklmnopqrstuvwxyz"
```

```
/opt/stonesoup/workspace/testData/bad02.txt


Where the above is a file containing the text "this string is really long and does
not contain a null char soon enough"
```

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Memory Corruption
- Restrictions and Requirements

C - CWE-415A - Double Free

Summary

This snippet implements a heap allocated buffer that erroneously gets double `free()`'d causing a segfault. The snippet takes an input string and copies it into a heap allocated buffer. It then checks to see if the first character is an 'a' or greater and if so, calls a function that finishes by `free()`ing the buffer. The snippet then finishes execution and `free()`s the buffer, causing a double free if the first character was an 'a' or greater.

Weakness Class	Memory Corruption
CWE	CWE-415
Variant	A
Language	C
Status	WEAK-177 - CWE-415-0-C DONE
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

The snippet allocates a character array one size larger than the input string on the heap and copies the input string into it.

```
char *stonesoup_buffer = 0;
stonesoup_buffer = malloc((strlen(STONESOUP_TAINT_SOURCE) + 1) * sizeof(char ));
if (stonesoup_buffer == 0) {
    stonesoup_printf("Error: Failed to allocate memory\n");
    exit(1);
}
strcpy(stonesoup_buffer,STONESOUP_TAINT_SOURCE);
```

The snippet then checks the first character of the input string and if it is greater than or equal to 97 calls `stonesoup_process_buffer()` which in turn calls free() on the array allocated above. This is the crossover point.

```
/* STONESOUP: CROSSOVER-POINT (Improper Null Termination) */
read(stonesoup_file_desc,stonesoup_input_buf,128);
```

The snippet then proceeds to copy this text into the first character array by calling strcpy. If the second array does not contain a null terminator, strcpy will copy the string, and all the memory after it up to the first null byte, and put this into the first array overflowing the buffer and overwriting random memory. This is the trigger point and can result in a segfault.

```
/* STONESOUP: CROSSOVER-POINT (Double Free) */
if (stonesoup_buffer[0] >= 97) {
    stonesoup_printf("Index of first char:
%i\n",stonesoup_process_buffer(stonesoup_buffer));
}
```

```
char stonesoup_process_buffer(char *buffer_param)
{
  char first_char;
  first_char = buffer_param[0] - 97;
  free(buffer_param);
  return first_char;
}
```

The snippet then finishes execution and `free()`s the array allocated above. If this array has already been `free()`'d at the crossover point then this is a double free and will result in a segfault. This is the trigger point.

```
/* STONESOUP: TRIGGER-POINT (Double Free) */
free(stonesoup_buffer);
```

If the array had not been previously `free()`'d then the snippet will complete execution without error.

Benign Inputs

Any string starting with an ASCII character who's value is 96 or lower is benign. For example:

```
AAA
```

```
12345 Hello world!
```

```
[Good-bye cruel world!]
```

Exploiting Inputs

Any string starting with an ASCII character who's value is 97 or greater is exploiting. For example:

```
aaa
```

```
this string begins with a lower-case letter
```

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment

- Memory Corruption
- Restrictions and Requirements

C - CWE-416A - Use After Free

Summary

This snippet implements a buffer that is used after it has been `free()`'d and it's memory allocated to another task. The snippet takes a string as input and copies this string into an internal buffer allocated on the heap. For certain inputs (any string starting with an ASCII character with value of 'a' or higher) the buffer is then `free()`'d and it's memory is re allocated to a pointer to a buffer, which in tern is set to point to the input string. The original buffer is then filled with the input string again, overwriting the pointer if the memory had previously been `free()`'d. When this pointer is later accessed, if the use after free has occurred, a segfault will occur due to the corrupted memory the pointer references.

Weakness Class	Memory Corruption
CWE	CWE-416
Variant	A
Language	C
Status	**WEAK-182** - CWE-416-0-C **DONE**
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

The snippet allocates character array of size 65528 on the stack and copies the input string into the array.

```
char *stonesoup_buffer = 0;
stonesoup_buffer_len = 4;
stonesoup_buffer = malloc(65528);
if (stonesoup_buffer != NULL) {
    strncpy(stonesoup_buffer, STONESOUP_TAINT_SOURCE, stonesoup_buffer_len);
```

The snippet then checks the first character of the input string, and if it is greater than or equal to 'a' calls a function that `free()`s the character array. The snippet then allocates 65528 bytes for a char **. If the original array was `free()`'d then it is almost guaranteed that the same chunk of memory will be allocated to the char ** since it is such a large block. This is the crossover point.

```
/* STONESOUP: CROSSOVER-POINT (Use After Free) */
if (stonesoup_buffer[0] >= 97) {
    stonesoup_main_first_char = stonesoup_process_buffer(stonesoup_buffer);
}
stonesoup_buffer_ptr = malloc(65528);
```

The snippet then has the pointer point to the original input string and copies the data from the input string into the original character array. If the array has previously been `free()`'d then the data copied into the array will overwrite the pointer to the input string (since they are the same block of memory) causing a segfault when the pointer is dereferenced.

```
*stonesoup_buffer_ptr = STONESOUP_TAINT_SOURCE;
/* STONESOUP: TRIGGER-POINT (Use After Free) */
strncpy(stonesoup_buffer, STONESOUP_TAINT_SOURCE, stonesoup_buffer_len);
stonesoup_buffer[stonesoup_buffer_len - 1] = '\0';
stonesoup_tainted_len = strlen( *stonesoup_buffer_ptr); /* fail*/
```

If the array has not been `free()`'d previously, a use after free error will not occur and the snippet will complete without error.

Benign Inputs

Any string starting with an ASCII character who's value is 96 or lower is benign. For example:

```
AAA
```

```
12345 Hello world!
```

```
[Good-bye cruel world!]
```

Exploiting Inputs

Any string starting with an ASCII character who's value is 97 or greater is exploiting. For example:

```
aaa
```

```
this string begins with a lower-case letter
```

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Memory Corruption
- Restrictions and Requirements

C - CWE-590A - Free of Memory not on the Heap

Summary

This snippet implements stack allocated buffer which, under certain inputs, gets erroneously `free()`'d. The snippet takes a string as input and copies it to an internal array of size 64 allocated on the stack. This string is the converted to all caps and, if the resulting string's first letter

is an 'O' or greater, calls free on the stack allocated buffer, resulting in a segfault.

Weakness Class	Memory Corruption
CWE	CWE-590
Variant	A
Language	C
Status	WEAK-187 - CWE-590-0-C DONE
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

The snippet allocates character array of size 64 on the stack and copies the input string into this array. The snippet then iterates through the array converting it to capitals.

```
char stonesoup_function_buff[64];
strncpy(stonesoup_function_buff,STONESOUP_TAINT_SOURCE,63);
stonesoup_function_buff[63] = '\0';
for (; stonesoup_oc_i < 64; ++stonesoup_oc_i) {
    stonesoup_function_buff[stonesoup_oc_i] =
stonesoup_toupper(stonesoup_function_buff[stonesoup_oc_i]);
}
```

The snippet then calls a function which checks the first character in the modified string and if this character is 'O' or greater, erroneously calls free() on the stack allocated buffer causing a segfault. This is both the crossover and the trigger point.

```
void stonesoup_free_func(char *buff) {
  if (buff[0] >= 79) {
    /* STONESOUP: CROSSOVER-POINT (Free of Invalid Pointer Not on the Heap) */
    /* STONESOUP: TRIGGER-POINT (Free of Invalid Pointer Not on the Heap) */
    free(buff);
  }
}
```

If the first character is less than an 'O' the snippet will finish execution without error.

Benign Inputs

Any string starting with an ASCII character who's value is lower than 79 or between 96 and 110 is benign. For example:

```
AAAAAAAAAAAAAAAAAAAAAAA
```

```
Hello World!
```

```
many characters are good, but most are bad
```

Exploiting Inputs

Any string starting with an ASCII character who's value is between 79 and 95 or greater than 110 is exploiting. For example:

```
starting with an s is bad
```

```
zebras have stripes
```

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Memory Corruption
- Restrictions and Requirements

C - CWE-590B - Free of Memory not on the Heap

Summary

This snippet implements stack allocated buffer which, under certain inputs, gets erroneously `free()`'d. The snippet takes a string as input and copies it to an internal array of size 64 allocated on the stack in a global scope. This string is the converted to all caps and, if the resulting string's first letter is an 'O' or greater, calls free on the stack allocated buffer, resulting in a segfault.

Weakness Class	Memory Corruption
CWE	CWE-590
Variant	A
Language	C
Status	**WEAK-190** - CWE-590-1-C **DONE**
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

The snippet allocates character array of size 64 on the stack as a global variable and copies the input string into this array. The snippet then iterates through the array converting it to capitals.

```
char stonesoup_function_buff[64];

{...}
    strncpy(stonesoup_function_buff,STONESOUP_TAINT_SOURCE,63);
    stonesoup_function_buff[63] = '\0';
    for (; stonesoup_oc_i < 64; ++stonesoup_oc_i) {
        stonesoup_function_buff[stonesoup_oc_i] =
stonesoup_toupper(stonesoup_function_buff[stonesoup_oc_i]);
    }
```

The snippet then calls a function which checks the first character in the modified string and if this character is 'O' or greater, erroneously calls free() on the stack allocated buffer causing a segfault. This is both the crossover and the trigger point.

```
void stonesoup_free_func(char *buff) {
  if (buff[0] >= 79) {
    /* STONESOUP: CROSSOVER-POINT (Free of Invalid Pointer Not on the Heap) */
    /* STONESOUP: TRIGGER-POINT (Free of Invalid Pointer Not on the Heap) */
    free(buff);
  }
}
```

If the first character is less than an 'O' the snippet will finish execution without error.

Benign Inputs

Any string starting with an ASCII character who's value is lower than 79 or between 96 and 110 is benign. For example:

```
AAAAAAAAAAAAAAAAAAAAAAA
```

```
Hello World!
```

```
many characters are good, but most are bad
```

Exploiting Inputs

Any string starting with an ASCII character who's value is between 79 and 95 or greater than 110 is exploiting. For example:

```
starting with an s is bad
```

```
zebras have stripes
```

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Memory Corruption
- Restrictions and Requirements

C - CWE-761A - Free of Memory not at Start of Buffer

Summary

This snippet allocates a buffer on the heap, copies the input string into it, and then capitalizes each letter in the buffer. It searches that buffer to see if it contains the letter 'E', using a while loop that increments the pointer to the buffer each time through the loop. When it finds a letter 'E', or reaches the end of the buffer, it frees the buffer, using the pointer that it may have incremented. If the input is the empty string, or has the letter 'E' (or 'e') as its first character, the free happens successfully. If the input is non-empty and does not have the letter 'E' as its first character, the free does not happen at the start of the buffer, resulting in a segfault.

Weakness Class	Memory Corruption
CWE	CWE-761
Variant	A
Language	C
Status	WEAK-196 - CWE-761-0-C DONE
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

The snippet allocates on the heap a character array with size equal to one more than the length of the source taint plus one.

```
stonesoup_buffer = malloc(stonesoup_buffer_len * sizeof(char ));
```

The snippet then copies the soure taint into this heap-allocated buffer.

```
strcpy(stonesoup_buffer,STONESOUP_TAINT_SOURCE);
```

The snippet steps through stonesoup_buffer, capitalizing each letter:

```
for (; stonesoup_oc_i < stonesoup_buffer_len; ++stonesoup_oc_i) {
    stonesoup_buffer[stonesoup_oc_i] =
stonesoup_toupper(stonesoup_buffer[stonesoup_oc_i]);
}
```

The snippet then calls the function stonesoup_contains_char, using the heap-allocated buffer and the character 'E'.

```
stonesoup_found = stonesoup_contains_char(stonesoup_buffer,'E');
```

The stonesoup_contains_char function has the following signature:

```
int stonesoup_contains_char(char *str_param,char c_param);
```

It intends to check if str_param contains the character c_param, returning 1 if it does contain it and 0 otherwise, then free the buffer. However, it has a bug.

stonesoup_contains_char steps through the str_param buffer in a while loop, incrementing str_param by one each time through the loop. If the character pointed to by str_param is the same as c_param, it sets function_found to 1 and breaks from the loop. This loop is the crossover point.

```
/* STONESOUP: CROSSOVER-POINT (Free Not At Start Of Buffer) */
while( *str_param != 0){
  if ( *str_param == c_param) {
    function_found = 1;
    break;
  }
  str_param = str_param + 1;
}
```

After the loop executes, stonesoup_contains_char frees str_param. This is the trigger point.

```
free(str_param);
```

If the input begins with the letter 'E' (or 'e'), then str_param will still point to the front of the buffer, and the free command will execute correctly. If the input does not begin with the letter 'E' (or 'e'), then str_param points to somewhere else in the buffer (or the terminating null, and the free operation will cause a crash.

Benign Inputs

Any string beginning with the letter 'E' or 'e', or the empty string. For example:

```
eAAA
```

```
e12345 Hello world!
```

```
e This string begins with an e
```

Exploiting Inputs

Any non-empty string that does not start with a letter 'e' or 'E'. For example:

```
aaa
```

```
this string has doe not begin with an e
```

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Memory Corruption
- Restrictions and Requirements

C - CWE-785A - Use of Path Manipulation Function without Maximum-sized Buffer on the Stack

Summary

This snippet allocates a buffer on the stack, with a canary function pointer immediately after it. It checks if the taint source is less than 20 characters, and if so, it sets the canary function pointer to strlen and call realpath on the taint source, putting the result into the stack buffer. If realpath evaluates to 20 or more characters, realpath will overwrite the canary function pointer. The snippet then calls the canary function pointer. If this pointer has been over-written, this call will result in a segmentation fault or illegal instruction fault.

For weakness C-CWE-785A, the path manipulation occurs on the stack. It causes a problem when it overflows into a canary function pointer, and that canary function pointer is subsequently called.

For weakness C-CWE-785B, the path manipulation occurs on the heap. It causes a problem when it over-writes memory manager values or user-defined values on the heap.

For weakness C-CWE-785C, the path manipulation occurs on the heap, but it first overflows within a user-defined struct. This allows the weakness to have full control of the error behavior.

For weakness C-CWE-785D, the path manipulation occurs on the stack, but it first overflows within a user-defined struct. This allows the weakness to have full control of the error behavior.

Weakness Class	Memory Corruption
CWE	CWE-785
Variant	A
Language	C
Status	**WEAK-223** - CWE-785-0-C **DONE**
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

The snippet allocates on the stack a pointer to a canary function (stonesoup_canary_pointer), and a 20-character buffer (stonesoup_base_path). The canary function pointer is created as an array of size 1, rather than directly as a pointer, because the C compiler often moves arrays to be next to each other on the stack. If the canary function pointer was simply a pointer, it is likely to be allocated away from the 20-character buffer on the stack. To reliably trigger the weakness, the stonesoup_canary_pointer array should be directly next to the stonesoup_base_path buffer.

```
size_t (*stonesoup_canary_pointer[1])(const char *);
char stonesoup_base_path[20];
```

It checks if the taint source is less than 20 characters. If it is not, then it does nothing. If it is, it proceeds with the weakness.

It sets the canary pointer to be the C library function strlen, then sets the 20-character buffer to all 0's.

```
stonesoup_canary_pointer[0] = strlen;
memset(stonesoup_base_path,0,20);
```

The snippet calls realpath on the taint source, putting the result into stonesoup_base_path. If realpath evaluates to more than 20 characters, then stonesoup_base_path will overflow into the stonesoup_canary_pointer. This is both the crossover point and the trigger point, because the system may now be in an unintended state.

```
/* STONESOUP: CROSSOVER-POINT (Use of Path Manipulation Function without
Maximum-sized Buffer) */
/* STONESOUP: TRIGGER-POINT (Use of Path Manipulation Function without Maximum-sized
Buffer) */
realpath(STONESOUP_TAINT_SOURCE,stonesoup_base_path);
```

The snippet then uses the stonesoup_canary_pointer function to determine the string length of stonesoup_base_path. If the stonesoup_base_path buffer has overflowed, the value of stonesoup_canary_pointer will have been over-written, and stonesoup_canary_pointer is unlikely to be a valid function pointer. This function call will then result in a segmentation fault or illegal instruction fault.

```
stonesoup_opt_var = stonesoup_canary_pointer[0](stonesoup_base_path);
```

Benign Inputs

Any string that is 20 characters or more (because the weakness will not execute), or that is less than 20 characters and whose realpath also evaluates to less than 20 characters. For example:

```
/etc/passwd
```

```
/etc/ssh/ssh_config
```

```
/usr/bin/gcc
```

Exploiting Inputs

Any string that is less than 20 characters, but whose realpath evaluates to 20 characters or more. For example:

```
../testData
```

Since the code runs from /opt/stonesoup/workspace/testData, realpath on this string will evaluate to /opt/stonesoup/workspace/testData. This string has 34 characters (35 with the terminating null), and will over-write the canary function pointer.

```
../testOutput
```

Since the code runs from /opt/stonesoup/workspace/testData, realpath on this string will evaluate to /opt/stonesoup/workspace/testOutput. This string has 36 characters (37 with the terminating null), and will over-write the canary function pointer.

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Memory Corruption
- Restrictions and Requirements

C - CWE-785B - Use of Path Manipulation Function without Maximum-sized Buffer on the Heap

Summary

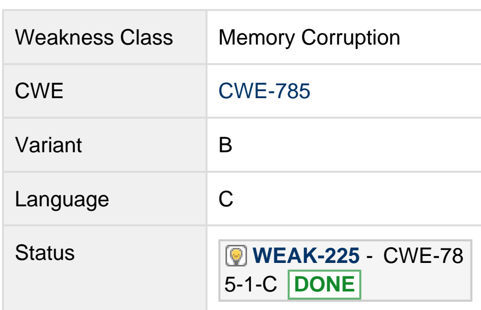
This snippet checks if the taint source is less than 20 characters, and if so, allocates a buffer on the heap with 20 characters. It sets the buffer to all 0's, then calls realpath on the taint source, with the destination being the 20-character heap buffer. If realpath evaluates to more than 20 characters, it will over-write memory reserved for the memory manager, resulting in a glibc error when the heap buffer is freed.

For weakness C-CWE-785A, the path manipulation occurs on the stack. It causes a problem when it overflows into a canary function pointer, and that canary function pointer is subsequently called.

For weakness C-CWE-785B, the path manipulation occurs on the heap. It causes a problem when it over-writes memory manager values or user-defined values on the heap.

For weakness C-CWE-785C, the path manipulation occurs on the heap, but it first overflows within a user-defined struct. This allows the weakness to have full control of the error behavior.

For weakness C-CWE-785D, the path manipulation occurs on the stack, but it first overflows within a user-defined struct. This allows the weakness to have full control of the error behavior.



Weakness Class	Memory Corruption
CWE	CWE-785
Variant	B
Language	C
Status	**WEAK-225** - CWE-785-1-C DONE

Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

The snippet checks if the taint source is less than 20 characters. If it is, it allocates a 20-character buffer on the heap.

```
stonesoup_buff = (char *) malloc (sizeof(char) * 20);
```

It then sets this buffer to all 0's.

```
memset(stonesoup_buff, 0, 20);
```

The snippet calls realpath on the taint source, putting the result into stonesoup_buff. If realpath evaluates to more than 20 characters, then stonesoup_buff will overflow into the reserved space for the memory manager. This is both the crossover point and the trigger point, because the system may now be in an unintended state.

```
/* STONESOUP: CROSSOVER-POINT (Use of Path Manipulation Function without
Maximum-sized Buffer) */
/* STONESOUP: TRIGGER-POINT (Use of Path Manipulation Function without Maximum-sized
Buffer) */
realpath(STONESOUP_TAINT_SOURCE, stonesoup_buff);
```

At the end of the weakness, the snippet frees stonesoup_buff. If stonesoup_buff has overflowed into the reserved space for the memory manager, the result will be a glibc error. To reliably see this failure, the MALLOC_CHECK_ environment variable must be set to 1.

```
free (stonesoup_buff);
```

Benign Inputs

Any string that is 20 characters or more (because the weakness will not execute), or that is less than 20 characters and whose realpath also evaluates to less than 20 characters. For example:

```
/etc/passwd
```

```
/etc/ssh/ssh_config
```

```
/usr/bin/gcc
```

Exploiting Inputs

Any string that is less than 20 characters, but whose realpath evaluates to 20 characters or more. For example:

```
../testData
```

Since the code runs from /opt/stonesoup/workspace/testData, realpath on this string will evaluate to /opt/stonesoup/workspace/testData. This string has 34 characters (35 with the terminating null), and will over-write the canary function pointer.

```
../testOutput
```

Since the code runs from /opt/stonesoup/workspace/testData, realpath on this string will evaluate to /opt/stonesoup/workspace/testOutput. This string has 36 characters (37 with the terminating null), and will over-write the canary function pointer.

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Memory Corruption
- Restrictions and Requirements

C - CWE-785C - Use of Path Manipulation Function without Maximum-sized Buffer on the Heap in a Struct

Summary

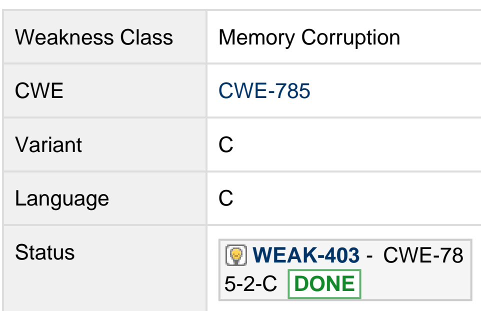
This snippet allocates a struct on the stack, with a 20-character buffer and a pointer following that buffer. It checks if the taint source is less than 20 characters. If so, it sets the 20-charcter buffer to all 0's then sets the pointer following the buffer to point to the beginning of the buffer. It then calls realpath with the destination being the 20-character buffer inside the struct. If realpath evaluates to more than 20 characters, the buffer inside the struct will overflow into the pointer inside the struct. The snippet then calls strlen on the pointer inside the struct. If an overflow has occurred, this pointer will be invalid, and a segmentation fault will occur.

For weakness C-CWE-785A, the path manipulation occurs on the stack. It causes a problem when it overflows into a canary function pointer, and that canary function pointer is subsequently called.

For weakness C-CWE-785B, the path manipulation occurs on the heap. It causes a problem when it over-writes memory manager values or user-defined values on the heap.

For weakness C-CWE-785C, the path manipulation occurs on the heap, but it first overflows within a user-defined struct. This allows the weakness to have full control of the error behavior.

For weakness C-CWE-785D, the path manipulation occurs on the stack, but it first overflows within a user-defined struct. This allows the weakness to have full control of the error behavior.



Weakness Class	Memory Corruption
CWE	CWE-785
Variant	C
Language	C
Status	WEAK-403 - CWE-785-2-C DONE

Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

The snippet checks if the taint source is less than 20 characters. If it is, it allocates on the heap a stonesoup_struct pointed to by stonesoup_data.

```
stonesoup_data = (struct stonesoup_struct*) malloc (sizeof(struct
stonesoup_struct));
```

stonesoup_struct has the following definition:

```
struct stonesoup_struct {
    char base_path[20];
    char * buff_pointer;
};
```

The snippet sets the stonesoup_data->base_path buffer to all 0's, then sets the stonesoup_data->buff_pointer to point to the beginning of stonesoup_data->base_path.

```
memset(stonesoup_data->base_path, 0, 20);
stonesoup_data->buff_pointer = stonesoup_data->base_path;
```

The snippet calls realpath on the taint source, putting the result into stonesoup_data->base_path. If realpath evaluates to more than 20 characters, then stonesoup_data->base_path will overflow into stonesoup_data->buff_pointer. This is both the crossover point and the trigger point, because the system may now be in an unintended state.

```
/* STONESOUP: CROSSOVER-POINT (Use of Path Manipulation Function without
Maximum-sized Buffer) */
/* STONESOUP: TRIGGER-POINT (Use of Path Manipulation Function without Maximum-sized
Buffer) */
realpath(STONESOUP_TAINT_SOURCE, stonesoup_data->base_path);
```

The snippet then uses calls strlen on stonesoup_data->buff_pointer. If the realpath call caused an overflow, stonesoup_data->buff_pointer will no longer be a valid pointer, and this operation will cause a segmentation fault.

```
stonesoup_opt_var = strlen( stonesoup_data->buff_pointer);
```

Benign Inputs

Any string that is 20 characters or more (because the weakness will not execute), or that is less than 20 characters and whose realpath also evaluates to less than 20 characters. For example:

```
/etc/passwd
```

```
/etc/ssh/ssh_config
```

```
/usr/bin/gcc
```

Exploiting Inputs

Any string that is less than 20 characters, but whose realpath evaluates to 20 characters or more. For example:

```
../testData
```

Since the code runs from /opt/stonesoup/workspace/testData, realpath on this string will evaluate to /opt/stonesoup/workspace/testData. This string has 34 characters (35 with the terminating null), and will over-write the canary function pointer.

```
../testOutput
```

Since the code runs from /opt/stonesoup/workspace/testData, realpath on this string will evaluate to /opt/stonesoup/workspace/testOutput. This string has 36 characters (37 with the terminating null), and will over-write the canary function pointer.

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Memory Corruption
- Restrictions and Requirements

C - CWE-785D - Use of Path Manipulation Function without Maximum-sized Buffer on the Stack in a Struct

Summary

This snippet allocates a struct on the stack, with a 20-character buffer and a pointer following that buffer. It checks if the taint source is less than 20 characters. If so, it sets the 20-charcter buffer to all 0's then sets the pointer following the buffer to point to the beginning of the buffer. It then calls realpath with the destination being the 20-character buffer inside the struct. If realpath evaluates to more than 20 characters, the buffer inside the struct will overflow into the pointer inside the struct. The snippet then calls strlen on the pointer inside the struct. If an overflow has occurred, this pointer will be invalid, and a segmentation fault will occur.

For weakness C-CWE-785A, the path manipulation occurs on the stack. It causes a problem when it overflows into a canary function pointer, and that canary function pointer is subsequently called.

For weakness C-CWE-785B, the path manipulation occurs on the heap. It causes a problem when it over-writes memory manager values or user-defined values on the heap.

For weakness C-CWE-785C, the path manipulation occurs on the heap, but it first overflows within a user-defined struct. This allows the weakness to have full control of the error behavior.

For weakness C-CWE-785D, the path manipulation occurs on the stack, but it first overflows within a user-defined struct. This allows the weakness to have full control of the error behavior.

Weakness Class	Memory Corruption
CWE	CWE-785
Variant	D
Language	C
Status	WEAK-403 - CWE-785-2-C DONE
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

The snippet allocates on the stack a stonesoup_struct called stonesoup_data.

```
struct stonesoup_struct stonesoup_data;
```

stonesoup_struct has the following definition:

```
struct stonesoup_struct {
    char base_path[20];
    char * buff_pointer;
};
```

The snippet checks if the taint source is less than 20 characters. If it is, it sets the base_path buffer to all 0's, then sets the buff_pointer field to point to the beginning of base_path.

```
for (stonesoup_i = 0; stonesoup_i < 20; stonesoup_i++) {
    stonesoup_data.base_path[stonesoup_i] = 0;
}
stonesoup_data.buff_pointer = stonesoup_data.base_path;
```

The snippet calls realpath on the taint source, putting the result into stonesoup_data.base_path. If realpath evaluates to more than 20 characters, then stonesoup_data.base_path will overflow into stonesoup_data.buff_pointer. This is both the crossover point and the trigger point, because the system may now be in an unintended state.

```
/* STONESOUP: CROSSOVER-POINT (Use of Path Manipulation Function without
Maximum-sized Buffer) */
/* STONESOUP: TRIGGER-POINT (Use of Path Manipulation Function without Maximum-sized
Buffer) */
realpath(STONESOUP_TAINT_SOURCE, stonesoup_data.base_path);
```

The snippet then uses calls strlen on stonesoup_data.buff_pointer. If the realpath call caused an overflow, stonesoup_data.buff_pointer will no longer be a valid pointer, and this operation will cause a segmentation fault.

```
stonesoup_opt_var = strlen( stonesoup_data.buff_pointer);
```

Benign Inputs

Any string that is 20 characters or more (because the weakness will not execute), or that is less than 20 characters and whose realpath also evaluates to less than 20 characters. For example:

```
/etc/passwd
```

```
/etc/ssh/ssh_config
```

```
/usr/bin/gcc
```

Exploiting Inputs

Any string that is less than 20 characters, but whose realpath evaluates to 20 characters or more. For example:

```
../testData
```

Since the code runs from /opt/stonesoup/workspace/testData, realpath on this string will evaluate to /opt/stonesoup/workspace/testData. This string has 34 characters (35 with the terminating null), and will over-write the canary function pointer.

```
../testOutput
```

Since the code runs from /opt/stonesoup/workspace/testData, realpath on this string will evaluate to /opt/stonesoup/workspace/testOutput. This string has 36 characters (37 with the terminating null), and will over-write the canary function pointer.

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Memory Corruption
- Restrictions and Requirements

C - CWE-805A - Buffer Access With Incorrect Length Value on the Stack

Summary

This snippet takes a buffer as input and copies it into another buffer. It then converts the new buffer to uppercase and prints it out. If the provided input is larger than the buffer it is being copied into, then this will result in a buffer overwrite due to access with an incorrect length. This is due to using the input length in the strncpy call used to copy the buffer.

For weakness C-CWE-805A, the buffer access occurs on the stack. It causes a problem when it overwrites %eip, causing a segmentation fault at function return.

For weakness C-CWE-805B, the buffer access occurs on the heap. It causes a problem when it over-writes memory manager values or user-defined values on the heap.

For weakness C-CWE-805C, the buffer access occurs on the heap, but it first overflows within a user-defined struct. This allows the weakness to have full control of the error behavior.

For weakness C-CWE-805D, the buffer occurs on the stack, but it first overflows within a user-defined struct. This allows the weakness to have full control of the error behavior.

Weakness Class	Memory Corruption
CWE	CWE-805
Variant	A
Language	C
Status	WEAK-240 - CWE-805-0-C DONE
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

The snippet allocates a character array of a small size.

```
char stonesoup_buffer[8];
```

It then copies the input buffer into the allocated buffer. If the input buffer is larger than the allocated buffer, this will result in a buffer overflow.

```
/* STONESOUP: CROSSOVER-POINT (Buffer Access With Incorrect Length Value) */
/* STONESOUP: TRIGGER-POINT (Buffer Access With Incorrect Length Value) */
strncpy(stonesoup_buffer,STONESOUP_TAINT_SOURCE,strlen(STONESOUP_TAINT_SOURCE) + 1);
```

Finally the snippet attempts to convert the copied buffer to uppercase and print it out. If a buffer overflow happened, then this will cause the program to crash.

```
for (; stonesoup_oc_i < strlen(stonesoup_buffer); ++stonesoup_oc_i) {
     stonesoup_buffer[stonesoup_oc_i] =
stonesoup_toupper(stonesoup_buffer[stonesoup_oc_i]);
}
stonesoup_printf("%s\n",stonesoup_buffer);
```

Benign Inputs

Any string input less than 8 characters. For example:

```
Hello
```

```
shortst
```

```
a
```

Exploiting Inputs

Any string input greater than 8 characters. For example:

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

```
zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz
zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz
zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz
zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz
zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz
zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz
zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz
zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz
zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz
zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz
zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz
zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz
zzzzzzzzzzzzzzz
```

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Memory Corruption
- Restrictions and Requirements

C - CWE-805B - Buffer Access With Incorrect Length Value on the Heap

Summary

This snippet takes a buffer as input and copies it into another buffer. It then converts the new buffer to uppercase and prints it out. If the provided input is larger than the buffer it is being copied into, then this will result in a buffer overwrite due to access with an incorrect length. This is due to using the input length in the strncpy call used to copy the buffer.

For weakness C-CWE-805A, the buffer access occurs on the stack. It causes a problem when it overwrites %eip, causing a segmentation fault at function return.

For weakness C-CWE-805B, the buffer access occurs on the heap. It causes a problem when it over-writes memory manager values or user-defined values on the heap.

For weakness C-CWE-805C, the buffer access occurs on the heap, but it first overflows within a user-defined struct. This allows the weakness to have full control of the error behavior.

For weakness C-CWE-805D, the buffer occurs on the stack, but it first overflows within a user-defined struct. This allows the weakness to have full control of the error behavior.

Weakness Class	Memory Corruption
CWE	CWE-805
Variant	B
Language	C
Status	WEAK-243 - CWE-805-1-C DONE
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

The snippet allocates a character array of a small size.

```
char * stonesoup_data = 0;
stonesoup_data = (char*) malloc(8 * sizeof(char));
```

It then copies the input buffer into the allocated buffer. If the input buffer is larger than the allocated buffer, this will result in a buffer overflow.

```
/* STONESOUP: CROSSOVER-POINT (Buffer Access With Incorrect Length Value) */
/* STONESOUP: TRIGGER-POINT (Buffer Access With Incorrect Length Value) */
strncpy(stonesoup_data, STONESOUP_TAINT_SOURCE, strlen(STONESOUP_TAINT_SOURCE) + 1);
```

Finally the snippet attempts to convert the copied buffer to uppercase and print it out. If a buffer overflow happened, then this will cause the program to crash.

```
for (; stonesoup_oc_i < strlen(stonesoup_data); ++stonesoup_oc_i) {
     stonesoup_data[stonesoup_oc_i] =
stonesoup_toupper(stonesoup_data[stonesoup_oc_i]);
}
stonesoup_printf("%s\n", stonesoup_data);
```

Benign Inputs

Any string input less than 8 characters. For example:

```
Hello
```

```
a
```

```
ok_str
```

Exploiting Inputs

Any string input greater than 8 characters. For example:

```
This_input_is_too_long_to_be_contained_in_buffer
```

```
abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz
```

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Memory Corruption
- Restrictions and Requirements

C - CWE-805C - Buffer Access With Incorrect Length Value on the Heap in a Struct

Summary

This snippet allocates a struct on the heap that contains an 8-character buffer, followed by a pointer. The pointer is set to point to the beginning of the 8-character buffer. The taint source is copied into the 8-character buffer, using strncpy, but the length is incorrectly capped at the length of the taint source rather than the length of the 8-character buffer. If the taint source is 8 characters or longer, the strncpy will overflow the 8-character buffer, over-writing the pointer that follows it. The snippet then call strlen on the following pointer, and if this point is not valid (due to the overflow), this will cause a segmentation fault.

For weakness C-CWE-805A, the buffer access occurs on the stack. It causes a problem when it overwrites %eip, causing a segmentation fault at function return.

For weakness C-CWE-805B, the buffer access occurs on the heap. It causes a problem when it over-writes memory manager values or user-defined values on the heap.

For weakness C-CWE-805C, the buffer access occurs on the heap, but it first overflows within a user-defined struct. This allows the weakness to have full control of the error behavior.

For weakness C-CWE-805D, the buffer occurs on the stack, but it first overflows within a user-defined struct. This allows the weakness to have full control of the error behavior.

Weakness Class	Memory Corruption
CWE	CWE-805
Variant	C
Language	C
Status	WEAK-404 - CWE-805-2-C DONE
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

The snippet allocates on the heap a stonesoup_struct.

```
stonesoup_data = (struct stonesoup_struct*) malloc(sizeof (struct
stonesoup_struct));
```

stonesoup_struct is defined as follows:

```
struct stonesoup_struct {
    char buffer[8];
    char * buff_pointer;
};
```

The snippet sets stonesoup_data->buff_pointer to point to the beginning of stonesoup_data->buffer.

```
stonesoup_data->buff_pointer = stonesoup_data->buffer;
```

It then copies the taint source into stonesoup_data->buffer, but incorrectly uses the length of the taint source as the maximum number of character to copy, rather than the length of stonesoup_data->buffer (which is 8).

```
/* STONESOUP: CROSSOVER-POINT (Buffer Access With Incorrect Length Value) */
/* STONESOUP: TRIGGER-POINT (Buffer Access With Incorrect Length Value) */
strncpy(stonesoup_data->buffer, STONESOUP_TAINT_SOURCE,
strlen(STONESOUP_TAINT_SOURCE) + 1);
```

If the taint source is 8 character of longer, this copy will overflow into stonesoup_data->buff_pointer.

Finally the snippet calls strlen on stonesoup_data->buff_pointer. If an overflow has occurred, this will be an invalid pointer and the strlen call will cause a segmentation fault.

```
stonesoup_ptr_deref = strlen( stonesoup_data->buff_pointer);
```

Benign Inputs

Any string input less than 8 characters. For example:

```
Hello
```

```
a
```

```
ok_str
```

Exploiting Inputs

Any string input greater than 8 characters. For example:

```
This_input_is_too_long_to_be_contained_in_buffer
```

```
abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz
```

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Memory Corruption
- Restrictions and Requirements

C - CWE-805D - Buffer Access With Incorrect Length Value on the Stack in a Struct

Summary

This snippet allocates a struct on the heap that contains an 8-character buffer, followed by a pointer. The pointer is set to point to the beginning of the 8-character buffer. The taint source is copied into the 8-character buffer, using strncpy, but the length is incorrectly capped at the length of the taint source rather than the length of the 8-character buffer. If the taint source is 8 characters or longer, the strncpy will overflow the 8-character buffer, over-writing the pointer that follows it. The snippet then call strlen on the following pointer, and if this point is not valid (due to the overflow), this will cause a segmentation fault.

For weakness C-CWE-805A, the buffer access occurs on the stack. It causes a problem when it overwrites %eip, causing a segmentation fault at function return.

For weakness C-CWE-805B, the buffer access occurs on the heap. It causes a problem when it over-writes memory manager values or user-defined values on the heap.

For weakness C-CWE-805C, the buffer access occurs on the heap, but it first overflows within a user-defined struct. This allows the weakness to have full control of the error behavior.

For weakness C-CWE-805D, the buffer occurs on the stack, but it first overflows within a user-defined struct. This allows the weakness to

have full control of the error behavior.

Weakness Class	Memory Corruption
CWE	CWE-805
Variant	D
Language	C
Status	WEAK-424 - CWE-805-3-C DONE
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

The snippet allocates on the stack a stonesoup_struct.

```
struct stonesoup_struct stonesoup_data;
```

stonesoup_struct is defined as follows:

```
struct stonesoup_struct {
    char buffer[8];
    char * buff_pointer;
};
```

The snippet sets stonesoup_data,buff_pointer to point to the beginning of stonesoup_data.buffer.

```
stonesoup_data.buff_pointer = stonesoup_data.buffer;
```

It then copies the taint source into stonesoup_data.buffer, but incorrectly uses the length of the taint source as the maximum number of characters to copy, rather than the length of stonesoup_data.buffer (which is 8).

```
/* STONESOUP: CROSSOVER-POINT (Buffer Access With Incorrect Length Value) */
/* STONESOUP: TRIGGER-POINT (Buffer Access With Incorrect Length Value) */
strncpy(stonesoup_data.buffer, STONESOUP_TAINT_SOURCE,
strlen(STONESOUP_TAINT_SOURCE) + 1);
```

If the taint source is 8 character of longer, this copy will overflow into stonesoup_data.buff_pointer.

Finally the snippet calls strlen on stonesoup_data.buff_pointer. If an overflow has occurred, this will be an invalid pointer and the strlen call will cause a segmentation fault.

```
stonesoup_ptr_deref = strlen( stonesoup_data.buff_pointer);
```

Benign Inputs

Any string input less than 8 characters. For example:

```
Hello
```

```
a
```

```
ok_str
```

Exploiting Inputs

Any string input greater than 8 characters. For example:

```
This_input_is_too_long_to_be_contained_in_buffer
```

```
abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz
```

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Memory Corruption
- Restrictions and Requirements

C - CWE-806A - Buffer Access Using Size of Source Buffer on the Heap

Summary

This snippet creates two buffers on the stack, one of 64 bytes and one of 1024 bytes. It copies the taint source into the larger buffer. It checks if the length of the taint source is less than the length of the shorter buffer. If it is, it uses strncpy to copy the taint source into the shorter buffer, with a maximum value of 1024 bytes. However, strncpy always writes the maximum number of bytes, and so writes 1024 bytes. This overflows the short buffer, and corrupts other data on the stack. The %eip is corrupted, resulting in a segmentation fault at function return.

For weakness C-CWE-806A, the buffer access occurs on the stack. It causes a problem when it overwrites %eip, causing a segmentation fault at function return.

For weakness C-CWE-806B, the buffer access occurs on the heap. It causes a problem when it over-writes memory manager values or user-defined values on the heap.

For weakness C-CWE-806C, the buffer access occurs on the heap, but it first overflows within a user-defined struct. This allows the weakness to have full control of the error behavior.

For weakness C-CWE-806D, the buffer access occurs on the stack, but it first overflows within a user-defined struct. This allows the weakness to have full control of the error behavior.

This weakness differs from C-CWE-805A because in C-CWE-805A the strncpy uses the length of the taint source as its maximum copy

value, while in this weakness, the strncpy uses the length of the source buffer.

Weakness Class	Memory Corruption
CWE	CWE-806
Variant	A
Language	C
Status	WEAK-246 - CWE-806-0-C DONE
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

The snippet allocates two buffers on the stack. stonesoup_buff has size 64, and stonesoup_source has size 1024.

```
char stonesoup_buff[64];
char stonesoup_source[1024];
```

It sets stonesoup_buff to have all A's, with a terminating null, and it sets stonesoup_source to have all 0's.

```
memset(stonesoup_buff, 65, 64);
stonesoup_buff[63] = '\0';
memset(stonesoup_source,0,1024);
```

It then copies the taint source into stonesoup_source (the 1024-character buffer), using strncpy, with a maximum size of the stonesoup_source buffer. It places a null in the final character of stonesoup_source.

```
strncpy(stonesoup_source,STONESOUP_TAINT_SOURCE,sizeof(stonesoup_source));
stonesoup_source[1023] = 0;
```

It checks that the length of the string in stonesoup_source is less than the size of stonesoup_buff:

```
if (strlen(stonesoup_source) + 1 <= sizeof(stonesoup_buff))  {...}
```

If it is, it copies stonesoup_source into stonesoup_buff, but incorrectly uses the sizeof stonesoup_source as the maximum number of characters to copy, rather than the sizeof stonesoup_buff.

```
/* STONESOUP: CROSSOVER-POINT (Buffer Access Using Size of Source Buffer) */
/* STONESOUP: TRIGGER-POINT (Buffer Access Using Size of Source Buffer) */
strncpy(stonesoup_buff,stonesoup_source,sizeof(stonesoup_source));
```

strncpy pads the copied string with 0's until it reaches the length parameter. In this case, the string in stonesoup_source will be copied into stonesoup_buff, then 0's will be appended until it reaches the sizeof(stonesoup_source), which is 1024. This will overwrite %eip, resulting in a segmentation fault upon return.

If the length of the string in stonesoup_source is greater than the size of stonesoup_buff, then the if-body will not be executed, there will be no incorrect strncpy, and the behavior of the weakness will be benign.

Benign Inputs

Any string input 64 or more character long. For example:

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAaAAAAAAAAAA
AAAAAA
```

```
Hello World!  If this statement isn't more than 64 characters, it will go badly
```

```
Short is good for buffer overflows, unless you make a mistake and get it wrong
```

Exploiting Inputs

Any string input 63 characters or fewer. For example:

```
Short is now bad
```

```
AAAAAAAAAAAA
```

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Memory Corruption
- Restrictions and Requirements

C - CWE-806B - Buffer Access Using Size of Source Buffer on the Heap

Summary

This snippet creates a buffer on the stack of 1024 bytes and buffer on the heap of 64 bytes. It copies the taint source into the 1024-char buffer. It checks if the length of the taint source is less than the length of the 64-byte buffer. If it is, it uses strncpy to copy the taint source into the 64-byte buffer, with a maximum value of 1024 bytes. However, strncpy always writes the maximum number of bytes, and so writes 1024 bytes. This overflows the other data in the heap, including memory reserved for use by the memory manager. When the snippet attempts to free the heap-allocated buffer, a glibc error occurs.

For weakness C-CWE-806A, the buffer access occurs on the stack. It causes a problem when it overwrites %eip, causing a segmentation fault at function return.

For weakness C-CWE-806B, the buffer access occurs on the heap. It causes a problem when it over-writes memory manager values or user-defined values on the heap.

For weakness C-CWE-806C, the buffer access occurs on the heap, but it first overflows within a user-defined struct. This allows the weakness to have full control of the error behavior.

For weakness C-CWE-806D, the buffer access occurs on the stack, but it first overflows within a user-defined struct. This allows the weakness to have full control of the error behavior.

This weakness differs from C-CWE-805B because in C-CWE-805B the strncpy uses the length of the taint source as its maximum copy value, while in this weakness, the strncpy uses the length of the source buffer.

Weakness Class	Memory Corruption
CWE	CWE-806
Variant	B
Language	C
Status	WEAK-247 - CWE-806-1-C DONE
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

The snippet allocates a buffer on the stack with size 1024, and a pointer to characters.

```
char stonesoup_source[1024];
char * stonesoup_buffer;
```

It allocates a 64 character buffer on the heap and set stonesoup_buffer to point to it.

```
stonesoup_buffer = (char*) malloc (sizeof(char*) * 64);
```

It set stonesoup_source to have all 0's and sets stonesoup_buffer to have all A's with a terminating null.

```
memset(stonesoup_source, 0, 1024);
memset(stonesoup_buffer, 65, 64);
stonesoup_buffer[64 - 1] = '\0';
```

It then copies the taint source into stonesoup_source (the 1024-character buffer on the stack), using strncpy, with a maximum size of the stonesoup_source buffer. It places a null in the final character of stonesoup_source.

```
strncpy(stonesoup_source, STONESOUP_TAINT_SOURCE, sizeof(stonesoup_source));
stonesoup_source[1023] = 0;
```

It checks that the length of the string in stonesoup_source is less than the size of 64 characters (which is the size of the heap-allocated buffer stonesoup_buffer):

```
if (strlen(stonesoup_source) + 1 <= 64) {...}
```

If it is, it copies stonesoup_source into stonesoup_buff, but incorrectly uses the sizeof stonesoup_source as the maximum number of characters to copy, rather than the sizeof stonesoup_buff.

```
/* STONESOUP: CROSSOVER-POINT (Buffer Access Using Size of Source Buffer) */
/* STONESOUP: TRIGGER-POINT (Buffer Access Using Size of Source Buffer) */
strncpy(stonesoup_buffer ,stonesoup_source, sizeof(stonesoup_source));
```

strncpy pads the copied string with 0's until it reaches the length parameter. In this case, the string in stonesoup_source will be copied into stonesoup_buffer, then 0's will be appended until it reaches the sizeof(stonesoup_source), which is 1024. This will overwrite memory reserved for the memory manager, or other memory on the heap, resulting in unpredictable behavior. When stonesoup_buffer is freed, a glibc error will occur. For reliable errors, set the environment variable MALLOC_CHECK_ to 1 before executing this weakness.

```
free(stonesoup_buffer);
```

If the length of the string in stonesoup_source is greater than the size of stonesoup_buffer, then the if-body will not be executed, there will be no incorrect strncpy, and the behavior of the weakness will be benign.

Benign Inputs

Any string input 64 or more character long. For example:

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAaAAAAAAAAAA
AAAAAA
```

```
Hello World!  If this statement isn't more than 64 characters, it will go badly
```

```
Short is good for buffer overflows, unless you make a mistake and get it wrong
```

Exploiting Inputs

Any string input 63 characters or fewer. For example:

```
Short is now bad
```

```
AAAAAAAAAAAA
```

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Memory Corruption
- Restrictions and Requirements

C - CWE-806C - Buffer Access Using Size of Source Buffer on the Heap in a Struct

Summary

This snippet creates a buffer on the stack of 1024 bytes and a struct on the heap containing a 64-byte buffer and a pointer to the beginning of that buffer. It copies the taint source into the 1024-char buffer. It checks if the length of the taint source is less than the length of the 64-byte buffer. If it is, it uses strncpy to copy the taint source into the 64-byte buffer, with a maximum value of 1024 bytes. However, strncpy always writes the maximum number of bytes, and so writes 1024 bytes. This overflows the other data in the struct, including the pointer that originally pointed to the beginning of the 64-byte buffer. The snippet then calls strlen on that pointer in the struct. However, since the pointer has been corrupted by the overflow, this causes a segmentation fault.

For weakness C-CWE-806A, the buffer access occurs on the stack. It causes a problem when it overwrites %eip, causing a segmentation fault at function return.

For weakness C-CWE-806B, the buffer access occurs on the heap. It causes a problem when it over-writes memory manager values or user-defined values on the heap.

For weakness C-CWE-806C, the buffer access occurs on the heap, but it first overflows within a user-defined struct. This allows the weakness to have full control of the error behavior.

For weakness C-CWE-806D, the buffer access occurs on the stack, but it first overflows within a user-defined struct. This allows the weakness to have full control of the error behavior.

This weakness differs from C-CWE-805C because in C-CWE-805C the strncpy uses the length of the taint source as its maximum copy value, while in this weakness, the strncpy uses the length of the source buffer.

Weakness Class	Memory Corruption
CWE	CWE-806
Variant	C
Language	C
Status	**WEAK-405** - CWE-806-2-C **DONE**
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

The snippet allocates a buffer on the stack with size 1024, and a pointer to a stonesoup_struct.

```
char stonesoup_source[1024];
struct stonesoup_struct * stonesoup_data = 0;
```

It allocates a stonesoup_struct on the heap.

```
stonesoup_data = (struct stonesoup_struct *) malloc (sizeof(struct
stonesoup_struct));
```

The stonesoup_struct has the following definition:

```
struct stonesoup_struct {
    char buffer[64];
    char * buff_pointer;
};
```

It set stonesoup_source to have all 0's, sets stonesoup_data->buffer to have all A's with a terminating null, and sets stonesoup_data->buff_pointer to point to the beginning of stonesoup_data->buffer.

```
memset(stonesoup_source, 0, 1024);
memset(stonesoup_data->buffer, 65, 64);
stonesoup_data->buffer[64 - 1] = '\0';
stonesoup_data->buff_pointer = stonesoup_data->buffer;
```

It then copies the taint source into stonesoup_source (the 1024-character buffer on the stack), using strncpy, with a maximum size of the stonesoup_source buffer. It places a null in the final character of stonesoup_source.

```
strncpy(stonesoup_source, STONESOUP_TAINT_SOURCE, sizeof(stonesoup_source));
stonesoup_source[1023] = 0;
```

It checks that the length of the string in stonesoup_source has size less than 64 characters (which is the size of the buffer stonesoup_data->buffer, which is on the heap):

```
if (strlen(stonesoup_source) + 1 <= 64) {...}
```

If it is, it copies stonesoup_source into stonesoup_data->buffer, but incorrectly uses the sizeof stonesoup_source as the maximum number of characters to copy, rather than the sizeof stonesoup_data->buffer.

```
/* STONESOUP: CROSSOVER-POINT (Buffer Access Using Size of Source Buffer) */
/* STONESOUP: TRIGGER-POINT (Buffer Access Using Size of Source Buffer) */
strncpy(stonesoup_data->buffer, stonesoup_source, sizeof(stonesoup_source));
```

strncpy pads the copied string with 0's until it reaches the length parameter. In this case, the string in stonesoup_source will be copied into stonesoup_data->buffer, then 0's will be appended until it reaches the sizeof(stonesoup_source), which is 1024. This will overwrite memory in the heap-allocated struct, memory reserved for the memory manager, or other memory on the heap, resulting in unpredictable behavior.

The snippet then calls strlen on stonesoup_data->buff_pointer. However, stonesoup_data->buff_pointer has been over-written, and this call results in a segmentation fault.

```
stonesoup_opt_var = strlen( stonesoup_data->buff_pointer);
```

If the length of the string in stonesoup_source is greater than the size of stonesoup_data->buffer, then the if-body will not be executed, there will be no incorrect strncpy, and the behavior of the weakness will be benign.

Benign Inputs

Any string input 64 or more character long. For example:

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAaAAAAAAAAAAAAA
AAAAAA
```

```
Hello World!  If this statement isn't more than 64 characters, it will go badly
```

```
Short is good for buffer overflows, unless you make a mistake and get it wrong
```

Exploiting Inputs

Any string input 63 characters or fewer. For example:

```
Short is now bad
```

```
AAAAAAAAAAAA
```

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Memory Corruption
- Restrictions and Requirements

C - CWE-806D - Buffer Access Using Size of Source Buffer on the Stack in a Struct

Summary

This snippet creates a buffer on the stack of 1024 bytes and a struct on the stack containing a 64-byte buffer and a pointer to the beginning of that buffer. It copies the taint source into the 1024-char buffer. It checks if the length of the taint source is less than the length of the 64-byte buffer. If it is, it uses strncpy to copy the taint source into the 64-byte buffer, with a maximum value of 1024 bytes. However, strncpy always writes the maximum number of bytes, and so writes 1024 bytes. This overflows the other data in the struct, including the pointer that originally pointed to the beginning of the 64-byte buffer. The snippet then calls strlen on that pointer in the struct. However, since the pointer has been corrupted by the overflow, this causes a segmentation fault.

For weakness C-CWE-806A, the buffer access occurs on the stack. It causes a problem when it overwrites %eip, causing a segmentation fault at function return.

For weakness C-CWE-806B, the buffer access occurs on the heap. It causes a problem when it over-writes memory manager values or user-defined values on the heap.

For weakness C-CWE-806C, the buffer access occurs on the heap, but it first overflows within a user-defined struct. This allows the weakness to have full control of the error behavior.

For weakness C-CWE-806D, the buffer access occurs on the stack, but it first overflows within a user-defined struct. This allows the weakness to have full control of the error behavior.

This weakness differs from C-CWE-805D because in C-CWE-805D the strncpy uses the length of the taint source as its maximum copy value, while in this weakness, the strncpy uses the length of the source buffer.

Weakness Class	Memory Corruption

CWE	CWE-806
Variant	D
Language	C
Status	WEAK-425 - CWE-806-3-C DONE
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

The snippet allocates a buffer on the stack with size 1024 and also allocates a stonesoup_struct on the stack.

```
char stonesoup_source[1024];
struct stonesoup_struct stonesoup_data;
```

stonesoup_struct has the following definition:

```
struct stonesoup_struct {
    char buffer[64];
    char * buff_pointer;
};
```

It set stonesoup_source to have all 0's, sets stonesoup_data.buffer to have all A's with a terminating null, and sets stonesoup_data.buff_pointer to point to the beginning of stonesoup_data.buffer.

```
for (stonesoup_i = 0; stonesoup_i < 1024; stonesoup_i++) {
    stonesoup_source[stonesoup_i] = 0;
}
for (stonesoup_i = 0; stonesoup_i < 64; stonesoup_i++) {
    stonesoup_data.buffer[stonesoup_i] = 65;
}
stonesoup_data.buffer[64 - 1] = '\0';
stonesoup_data.buff_pointer = stonesoup_data.buffer;
```

It then copies the taint source into stonesoup_source (the 1024-character buffer on the stack), using strncpy, with a maximum size of the stonesoup_source buffer. It places a null in the final character of stonesoup_source.

```
strncpy(stonesoup_source, STONESOUP_TAINT_SOURCE, sizeof(stonesoup_source));
stonesoup_source[1023] = 0;
```

It checks that the length of the string in stonesoup_source has size less than 64 characters (which is the size of the buffer stonesoup_data->buffer, which is on the heap):

```
if (strlen(stonesoup_source) + 1 <= 64) {...}
```

If it is, it copies stonesoup_source into stonesoup_data.buffer, but incorrectly uses the sizeof stonesoup_source as the maximum number of characters to copy, rather than the sizeof stonesoup_data.buffer.

```
/* STONESOUP: CROSSOVER-POINT (Buffer Access Using Size of Source Buffer) */
/* STONESOUP: TRIGGER-POINT (Buffer Access Using Size of Source Buffer) */
strncpy(stonesoup_data.buffer, stonesoup_source, sizeof(stonesoup_source));
```

strncpy pads the copied string with 0's until it reaches the length parameter. In this case, the string in stonesoup_source will be copied into stonesoup_data.buffer, then 0's will be appended until it reaches the sizeof(stonesoup_source), which is 1024. This will overwrite memory in the stack-allocated struct, other memory on the stack, or the return pointer for the function, resulting in unpredictable behavior.

The snippet then calls strlen on stonesoup_data.buff_pointer. However, stonesoup_data.buff_pointer has been over-written, and this call results in a segmentation fault.

```
stonesoup_opt_var = strlen( stonesoup_data.buff_pointer);
```

If the length of the string in stonesoup_source is greater than the size of stonesoup_data.buffer, then the if-body will not be executed, there will be no incorrect strncpy, and the behavior of the weakness will be benign.

Benign Inputs

Any string input 64 or more character long. For example:

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAaAAAAAAAAAA
AAAAAA
```

```
Hello World!  If this statement isn't more than 64 characters, it will go badly
```

```
Short is good for buffer overflows, unless you make a mistake and get it wrong
```

Exploiting Inputs

Any string input 63 characters or fewer. For example:

```
Short is now bad
```

```
AAAAAAAAAAAA
```

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment

- Memory Corruption
- Restrictions and Requirements

C - CWE-822A - Untrusted Pointer Dereference

Summary

This snippet creates a function pointer that takes 2 const char * as input and returns an int. If the length of the taint source is 0 mod 3, the snippet sets the function pointer to be strcmp. If the length of the taint source is 1 mod 3, the snippet sets the function pointer to strcoll. Otherwise the snippet sets the function pointer to be the address passed in by the taint source. When the snippet calls the function pointer, if the function pointer is strcmp or strcoll, the weakness proceeds without error. However, if the function pointer is the value passed in from the taint source, the function call results in a segmentation fault or illegal instruction fault.

Weakness Class	Memory Corruption
CWE	CWE-822
Variant	A
Language	C
Status	WEAK-251 - CWE-822-0-C DONE
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

For convenience, the snippet defines a stonesoup_fct_ptr as a function pointer that takes two strings and returns an int. This is the type signature of the C standard library function strcmp and strcoll.

```
typedef int (*stonesoup_fct_ptr)(const char *, const char *);
```

The snippet allocates a stonesoup_fct_ptr on the stack.

```
stonesoup_fct_ptr stonesoup_fp;
```

It then sets stonesoup_fp to the result of a function defined in the snippet:

```
stonesoup_fp = stonesoup_switch_func(STONESOUP_TAINT_SOURCE);
```

The function stonesoup_switch_func takes a string. It determines the value of the length of the string mod 3, and puts the result into var_len.

```
var_len = strlen(param) % 3;
```

If var_len is 0, it returns the C standard library function strcmp. If var_len is 1, it returns the C standard library function strcoll. These are both benign actions. If var_len is 2, the snippet scans the converts the parameter passed in into a pointer, and returns that pointer. This is an exploit behavior.

```
if (var_len == 0) {
    return strcmp;
  }
  else if (var_len == 1) {
    return strcoll;
  } else {
    sscanf(param,"%p",&fct_ptr_addr);
    return fct_ptr_addr;
  }
```

The snippet then uses stonesoup_fp to compare a random word (in this case criticisms_metallide) with the taint source.

```
stonesoup_cmp_flag = ( *stonesoup_fp)(stonesoup_rand_word,STONESOUP_TAINT_SOURCE);
```

If stonesoup_fp is strcmp or strcoll, the comparison happens without incident. Otherwise, the attempt to call stonesoup_fp will result in a segmentation fault or illegal instruction fault.

Benign Inputs

Any string whose length is 0 mod 3 or 1 mod 3.

```
abcdef
```

```
abc
```

```
abcdefghijklmnopqrstuvwxy
```

Exploiting Inputs

Any string whose length is 2 mod 3.

```
ab
```

```
ab000012
```

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment

- Memory Corruption
- Restrictions and Requirements

C - CWE-824A - Access of Uninitialized Pointer on the Stack

Summary

This snippet creates a struct on the stack that contains a function pointer and a char*. It examines the length of the taint source. If the length is not equal to 10, it sets the function pointer and char* within the struct to benign values. If the length is equal to 10, it does not set the function pointer or char* inside the struct, leaving them uninitialized. Subsequently, the snippet calls the function pointer from the struct, using the char* from the struct as an argument. If these values have not been initialized, this will result in a segmentation fault or illegal instruction fault.

For weakness C-CWE-824A, the uninitialized pointer is on the stack.

For weakness C-CWE-824B, the uninitialized pointer is on the heap.

Weakness Class	Memory Corruption
CWE	CWE-824
Variant	A
Language	C
Status	**WEAK-259** - CWE-824-0-C **DONE**
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

The snippet allocates a struct of type stonesoup_data_struct on the stack.

```
struct stonesoup_data_struct stonesoup_my_foo;
```

This struct is defined as follows:

```
struct stonesoup_data_struct {
  int (*func_member)(char *);
  char *str_member;
};
```

If the taint source is the empty string, the weakness exits without error. If the taint source is non-empty, the weakness calls the function stonesoup_set_function, with the taint source and the address of the struct stonesoup_my_foo.

```
stonesoup_set_function(STONESOUP_TAINT_SOURCE, &stonesoup_my_foo);
```

The function stonesoup_set_function takes a char* called set_param_str and a stonesoup_data_struct* called set_param_data_struct.

```
void stonesoup_set_function(char *set_param_str,struct stonesoup_data_struct
*set_param_data_struct);
```

If the length of set_param_str is greater than 10 or less than 10, it sets the values in set_param_data_struct to benign values.

stonesoup_modulus_function is a well-behaved function that takes a char* argument. It returns 0 if the ascii representation of the first character of the char* is even and 1 if it is odd.

```
if (strlen(set_param_str) > 10U) {
  set_param_data_struct -> func_member = stonesoup_modulus_function;
  set_param_data_struct -> str_member = set_param_str;
}
if (strlen(set_param_str) < 10U) {
  set_param_data_struct -> func_member = stonesoup_modulus_function;
  set_param_data_struct -> str_member = "default";
}
```

If the length of set_param_str is exactly 10, the function does not set any values in set_param_data_struct, and simply returns. In this case, the values of set_param_data_struct are not initialized.

Upon returning the the parent function, the snippet calls the function specified by stonesoup_my_foo.func_member, with the argument specified by stonesoup_my_foo.str_member.

```
stonesoup_val = (stonesoup_my_foo . func_member(stonesoup_my_foo . str_member));
```

If stonesoup_my_foo has been initialized, this operation is benign. If it has not been initialized, the operation will cause a segmentation fault or illegal instruction error.

Benign Inputs

Any string whose length is not equal to 10.

```
AAAAAAAAAAA
```

```
12345 Hello world!
```

```
This string does not have 10 characters
```

Exploiting Inputs

Any string whose length is exactly 10.

```
0123456789
```

```
10 chars..
```

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Memory Corruption
- Restrictions and Requirements

C - CWE-824B - Access of Uninitialized Pointer on the Heap

Summary

This snippet creates a struct on the stack that contains a function pointer and a char*. It examines the length of the taint source. If the length is not equal to 10, it sets the function pointer and char* within the struct to benign values. If the length is equal to 10, it does not set the function pointer or char* inside the struct, leaving them uninitialized. Subsequently, the snippet calls the function pointer from the struct, using the char* from the struct as an argument. If these values have not been initialized, this will result in a segmentation fault or illegal instruction fault.

For weakness C-CWE-824A, the uninitialized pointer is on the stack.

For weakness C-CWE-824B, the uninitialized pointer is on the heap.

Weakness Class	Memory Corruption
CWE	CWE-824
Variant	B
Language	C
Status	**WEAK-261** - CWE-824-1-C **DONE**
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

The snippet creates a function pointer of type fptr* on the stack.

```
fptr* stonesoup_function_ptr = 0;
```

fptr is defined as follows:

```
typedef int (*fptr)();
```

If the taint source is has 0 or 1 characters, the weakness exits without error. If the taint source has 2 or more characters, the weakness creates space for a function pointer on the heap.

```
stonesoup_function_ptr = malloc(sizeof(void *));
```

It then calls stonesoup_get_function with the length of the taint source and stonesoup_function_ptr.

```
stonesoup_get_function(stonesoup_input_len, stonesoup_function_ptr);
```

stonesoup_get_function has the following signature:

```
void stonesoup_get_function(int len, fptr * modulus_function);
```

This function sets the modulus_function (which is the value pointed to by stonesoup_function_ptr), based on the value of the len parameter. If len is greater or less than 10, *modulus_function is set to a benign function pointer that calculates the modulus:

```
if (len > 10) {
    *modulus_function = stonesoup_modulus_function1;
}
if (len < 10) {
    *modulus_function = stonesoup_modulus_function2;
}
```

If len is equal to 10, *modulus_function is not set and remains uninitialized.

Control returns to the parent function, which calls the function pointed to by stonesoup_function_ptr.

```
stonesoup_result = ( *stonesoup_function_ptr)(STONESOUP_TAINT_SOURCE);
```

If *stonesoup_function_ptr was not initialized, this will result in a segmentation fault or illegal instruction error.

Benign Inputs

Any string whose length is not equal to 10.

```
AAAAAAAAAAA
```

```
12345 Hello world!
```

```
This string does not have 10 characters
```

Exploiting Inputs

Any string whose length is exactly 10.

```
0123456789
```

```
10 chars..
```

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Memory Corruption
- Restrictions and Requirements

C - CWE-843A - Access of Resource Using Incompatible Type ('Type Confusion')

Summary

This snippet creates a struct that has a message_type field and a message_data field. The message_type field should be 1 if the message_data field contains a char*, and 0 if it contains an int. The message_data field is set to be the taint source, and the message_type field is set to 1. If the taint source can be converted to an integer, the message_data field is changed to the integer representation of the taint source. However, the message_type field is not updated. Subsequently, the snippet tries to calculate the string length of the message_data field. Since the message_data field is an int, rather than a char*, this causes a segmentation fault.

Weakness Class	Memory Corruption
CWE	CWE-843
Variant	A
Language	C
Status	WEAK-264 - CWE-843-0-C DONE
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

The snippet defines a struct stonesoup_message_buffer as follows:

```
struct stonesoup_message_buffer {
  union {
    int name_id_member;
    char *name_member;
  } message_data;
  int message_type;
};
```

This struct uses message_type to indicate whether the message_data field contains an int (name_id_member) or a char* (name_member).

The snippet allocates a stonesoup_message_buffer on the stack, then initializes the message_type to 1 and the message_data to the char* pointing to the taint source.

```
struct stonesoup_message_buffer stonesoup_buf;
stonesoup_buf . message_type = 1;
stonesoup_buf . message_data . name_member = STONESOUP_TAINT_SOURCE;
```

It attempts to convert the taint source to an integer, using the atoi function, and places the result in stonesoup_id.

```
stonesoup_id = atoi(STONESOUP_TAINT_SOURCE);
```

If stonesoup_id is not equal to 0, that is, if the taint source is a string containing an integer, it sets the message_data field of stonesoup_buf to be the integer stonesoup_id. However, it neglects to change the message_type of stonesoup_buf.

```
stonesoup_buf . message_data . name_id_member = stonesoup_id;
```

The snippet then calls the function stonesoup_process_buf on the struct stonesoup_buf. stonesoup_process_buf checks if the message_type in the struct is 1, indicating that the message_data in the struct is a char*. The message_type will always be 1, since the weakness never changes it. stonesoup_process_buf then takes the string length of the message_data parameter.

```
if (message_param -> message_type == 1)
  return strlen(message_param -> message_data . name_member);
else
  return message_param -> message_data . name_id_member;
```

If the message_data parameter is actually a name_id_member (that is, if the taint source can be converted to an integer), this will cause a segmentation fault.

Benign Inputs

Any string for which atoi returns 0, which is any string that does not represent a base 10 integer.

```
AAA
```

```
This is not a number
```

```
not a number not a number not a number
```

Exploiting Inputs

Any string that represents a base 10 integer.

```
012
```

```
40000
```

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Memory Corruption
- Restrictions and Requirements

Null Pointer Errors

This class contains weaknesses related to attempting to access null pointers.

Weakness Members

The number handling weakness class is defined as a set of weakness types as defined by the MITRE Common Weakness Enumeration (CWE) ontology. The MITRE CWE ontology is defined a a tree, and as such may target CWEs that are a parent or child of another CWE. In those cases, CWEs may be proposed to remove from the test set target given that they are fully covered by one or more other CWEs. Individual weakness variants (i.e. tests) are designed around the definition, description, and code examples provided for each CWE.

CWE	Name	Target Source Languages
CWE-476	NULL Pointer Dereference	C

There is only one weakness in the null pointer errors weakness class, so 7 number of different weakness snippets were developed to exercise a range of different cases.

Weakness Variants

The following weakness variants are developed and for the C source languages.

C Source Language

Error rendering macro 'detailssummary' : In template Confluence.Templates.Macro.MasterDetail.contentLink2: When evaluating "contextPath()": Error while computing function "contextPath()": null

Runtime Information

Notes

C - CWE-476A - NULL Pointer Dereference

Summary

This snippet reads entries from a comma-separated-value file. It expects to read 3 strings from a file in the format:

- double quote, up to 79 characters, double quote, comma
- double quote, up to 79 characters, double quote, comma
- double quote, up to 79 characters, double quote

The snippet then creates an array of 3 pointers, setting each pointer to NULL initially. It checks each string read from the file, and if the length of the string from the file is non-zero, it sets the corresponding pointer to that string. It then writes each of these three pointers to another file, using fputs.

If the file is not in the format expected, the three strings will not be read in properly. Some of the pointers in the array of 3 pointers will then not be set correctly and will remain NULL. When the snippet tries to write the strings to the output file, it will receive a segmentation fault on the fputs call.

In C-CWE476-A, the null pointer dereference arises because of mis-formatting in a comma-separated-values file.

In C-CWE476-B, the null pointer dereference arises because of poor logic in string searching.

In C-CWE476-C, the null pointer dereference arises because of mis-formatting in string parsing.

In C-CWE476-D, the null pointer dereference arises because of unexpected characters in an input string.

In C-CWE476-E, the null pointer dereference arises because of a file open on a file that does not exist.

In C-CWE476-F, the null pointer dereference arises because the function makes a call to a wrapped malloc, but the wrapped malloc has an incorrect implementation and fails to allocate space for a buffer.

In C-CWE476-G, the null pointer dereference arises because the function skipped a necessary call to allocate space for a buffer.

Weakness Class	Number Handling
CWE	CWE-476
Variant	A
Language	C
Status	WEAK-76 - CWE-476-0-C DONE
Negative Technical Impact(s)	DOS: Unconditional Exit

Implementation

The snippet creates three 80 character buffers on the stack.

```
char stonesoup_col1[80] = {0};
char stonesoup_col2[80] = {0};
char stonesoup_col3[80] = {0};
```

It also creates an array of three char* on the stack.

```
char *stonesoup_cols[3] = {0};
```

It then opens a file with the name specified by the taint source.

```
stonesoup_csv = fopen(STONESOUP_TAINT_SOURCE,"r");
```

If the file exists, it attempts to scan 3 values from the file, using fscanf.

```
fscanf(stonesoup_csv,"\"%79[^\"]\",\"%79[^\"]\",\"%79[^\"]\"",stonesoup_col1,stoneso
up_col2,stonesoup_col3);
```

The fscanf statement expects the file to contain information in the following format:

- double quote, up to 79 characters, double quote, comma
- double quote, up to 79 characters, double quote, comma
- double quote, up to 79 characters, double quote

If the file is improperly formatted, then some of the variables stonesoup_col1, stonesoup_col2, stonesoup_col3 will not be set.

In particular, if the first value is not preceded by a double quote, stonesoup_col1 will be all 0's. If stonesoup_col1 is read successfully, but stonesoup_col2 is not preceded by double quote, comma, double quote, stonesoup_col2 will be 0's. If stonesoup_col1 and stonesoup_col2 are read successfully, but stonesoup_col3 is not preceded by double quote, comma, double quote, stonesoup_col3 will be 0's.

The snippet then checks if it is has read the 3 values successfully. If the length of each string is greater than 0 (meaning that data has been read into it successfully), the snippet sets the corresponding entry of stonesoup_cols to be a pointer to that string. However, if the length of any of the strings is not greater than 0, the corresponding entry of stonesoup_cols remains 0.

```
if (strlen(stonesoup_col1) > 0)
    stonesoup_cols[0] = stonesoup_col1;
if (strlen(stonesoup_col2) > 0)
    stonesoup_cols[1] = stonesoup_col2;
if (strlen(stonesoup_col3) > 0)
    stonesoup_cols[2] = stonesoup_col3;
```

The snippet opens an output file, called "/opt/stonesoup/workspace/testData/myfile.txt".

```
stonesoup_temp = fopen(STONESOUP_EXPRESSION_1, "w+");
```

It attempts to write each of the three strings in stonesoup_cols to the output file.

```
fputs(stonesoup_cols[0],stonesoup_temp);
fputs(stonesoup_cols[1],stonesoup_temp);
fputs(stonesoup_cols[2],stonesoup_temp);
```

If any of the three strings are NULL, the corresponding fputs will cause a segmentation fault.

Benign Inputs

The name of a file a file that does not exist, or the name of a file containing 3 entries in the following format:

- double quote, up to 79 characters, double quote, comma
- double quote, up to 79 characters, double quote, comma
- double quote, up to 79 characters, double quote

For example:

```
/opt/stonesoup/workspace/testData/good01.csv

This file contains '"this","is","data"'
```

```
/opt/stonesoup/workspace/testData/good02.csv

This file contains '"hello","world","!!!!"'
```

```
/opt/stonesoup/workspace/testData/good03.csv

This file contains '"one","two","three"'
```

Exploiting Inputs

The name of a file that exists, but whose format does not match the specification above. For example:

```
/opt/stonesoup/workspace/testData/bad01.csv

This file contains '"this","is",data'
```

```
/opt/stonesoup/workspace/testData/bad02.csv

This file contains '"malformed values with spaces","because no",quotes'
```

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Restrictions and Requirements

C - CWE-476B - NULL Pointer Dereference

Summary

This snippet looks for the substring "aba" within the taint source. If it finds the substring, it sets a pointer called stonesoup_second_buff to the beginning of the "aba" substring, and the weakness continues without incident. If it does not find the substring, stonesoup_second_buff retains its initial value of NULL. The weakness subsequently calculates the length of stonesoup_second_buff, using strlen. If stonesoup_second_buff is NULL, this causes a segmentation fault.

In C-CWE476-A, the null pointer dereference arises because of mis-formatting in a comma-separated-values file.

In C-CWE476-B, the null pointer dereference arises because of poor logic in string searching.

In C-CWE476-C, the null pointer dereference arises because of mis-formatting in string parsing.

In C-CWE476-D, the null pointer dereference arises because of unexpected characters in an input string.

In C-CWE476-E, the null pointer dereference arises because of a file open on a file that does not exist.

In C-CWE476-F, the null pointer dereference arises because the function makes a call to a wrapped malloc, but the wrapped malloc has an incorrect implementation and fails to allocate space for a buffer.

In C-CWE476-G, the null pointer dereference arises because the function skipped a necessary call to allocate space for a buffer.

Weakness Class	Number Handling
CWE	CWE-476
Variant	B
Language	C
Status	WEAK-77 - CWE-476-1-C DONE
Negative Technical Impact(s)	DOS: Unconditional Exit

Implementation

The snippet creates a char* called stonesoup_second_buff and initializes it to NULL.

```
char *stonesoup_second_buff = 0;
```

It also creates a char* called stonesoup_finder, that is a substring it will search for in the taint source.

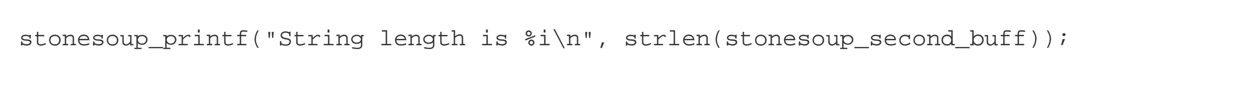
```
char *stonesoup_finder = "aba";
```

The snippet then loops through each character of the taint source, beginning with index 0 and going to index equal to the length of the taint source minus the length of the stonesoup_finder string.

At each index, it executes a second loop (with index stonesoup_j) to check if the stonesoup_finder string is contained within the taint source string. If it finds the stonesoup_finder string, it sets stonesoup_second_buff to the address of the beginning of the stonesoup_finder substring inside the taint source string.

```
for (stonesoup_i = 0;
     ((int )stonesoup_i) <= ((int )(strlen(STONESOUP_TAINT_SOURCE) -
strlen(stonesoup_finder)));
     ++stonesoup_i) {
    for (stonesoup_j = 0; stonesoup_j < strlen(stonesoup_finder); ++stonesoup_j) {
        if (STONESOUP_TAINT_SOURCE[stonesoup_i + stonesoup_j] !=
stonesoup_finder[stonesoup_j]) {
            stonesoup_check = 0;
            break;
        }
        stonesoup_check = 1;
    }
    /* STONESOUP: CROSSOVER-POINT (Null Pointer Dereference) */
    if (stonesoup_check == 1 && stonesoup_j == strlen(stonesoup_finder)) {
          stonesoup_printf("Found aba string\n");
          stonesoup_second_buff = &STONESOUP_TAINT_SOURCE[stonesoup_i];
          break;
    }
}
```

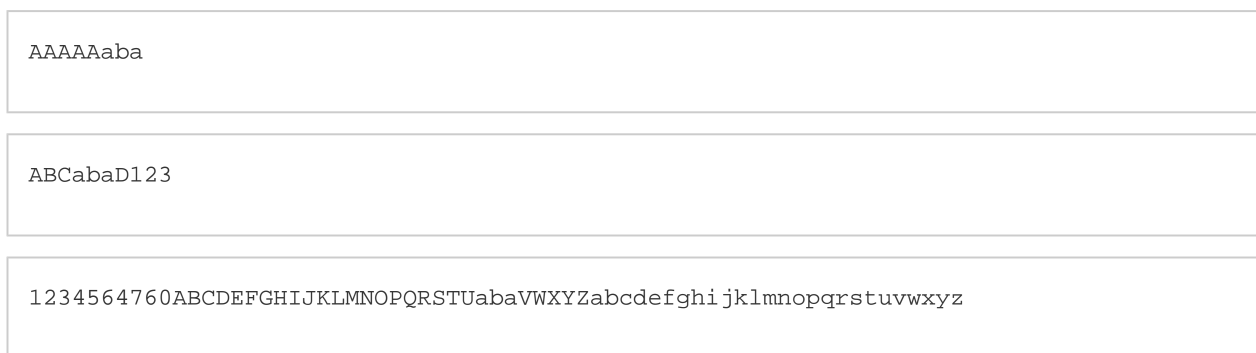
After the code exits the loop, it attempts to print the length of stonesoup_second_buff. If stonesoup_second_buff has not been set, the call to strlen will cause a segmentation fault.



```
stonesoup_printf("String length is %i\n", strlen(stonesoup_second_buff));
```

Benign Inputs

A string that contains the substring "aba". For example:



```
AAAAAaba
```

```
ABCabaD123
```

```
1234564760ABCDEFGHIJKLMNOPQRSTUabaVWXYZabcdefghijklmnopqrstuvwxyz
```

Exploiting Inputs

A string that does not contain the substring "aba". For example:

```
A.A
```

```
AAAAAAAHHHHHHHHHHHH!
```

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Restrictions and Requirements

C - CWE-476C - NULL Pointer Dereference

Summary

This snippet reads a space-delimited string from the taint source. The first element in the string is the number of elements following it. The snippets reads in the following elements and outputs them. If there are fewer elements than expected, a segmentation fault occurs.

In C-CWE476-A, the null pointer dereference arises because of mis-formatting in a comma-separated-values file.

In C-CWE476-B, the null pointer dereference arises because of poor logic in string searching.

In C-CWE476-C, the null pointer dereference arises because of mis-formatting in string parsing.

In C-CWE476-D, the null pointer dereference arises because of unexpected characters in an input string.

In C-CWE476-E, the null pointer dereference arises because of a file open on a file that does not exist.

In C-CWE476-F, the null pointer dereference arises because the function makes a call to a wrapped malloc, but the wrapped malloc has an incorrect implementation and fails to allocate space for a buffer.

In C-CWE476-G, the null pointer dereference arises because the function skipped a necessary call to allocate space for a buffer.

Weakness Class	Number Handling
CWE	CWE-476
Variant	C
Language	C
Status	**WEAK-81** - CWE-476-2-C **DONE**
Negative Technical Impact(s)	DOS: Unconditional Exit

Implementation

The snippet uses strtol to read an integer from the taint source. It puts the integer into stonesoup_len, and changes the pointer stonesoup_endptr to point to the first character after the first integer.

```
stonesoup_len = strtol(STONESOUP_TAINT_SOURCE,&stonesoup_endptr,10);
```

The snippet is expecting the taint source to begin with a number (stonesoup_len) followed by a space-delimited set of stonesoup_len strings.

The snippet next checks that the length is between 0 and 1000, to prevent too much memory from being consumed

```
if (stonesoup_len > 0 && stonesoup_len < 1000) {...}
```

It allocates space on the heap for an array of stonesoup_len pointers to characters, and then sets those pointers to 0.

```
stonesoup_values = malloc(stonesoup_len * sizeof(char *));
for (stonesoup_i = 0; stonesoup_i < stonesoup_len; ++stonesoup_i)
    stonesoup_values[stonesoup_i] = 0;
```

It loops stonesoup_len times, each time using sscanf to read a string into the 80-character buffer stonesoup_temp_ptr. It checks whether the return value of the sscanf is 1; if it is 1, the sscanf read a string successfully. If it did read a string successfully, it then allocates a buffer for a new copy of stonesoup_temp_str, puts the pointer to the newly allocated memory into the stonesoup_values array, and copies stonesoup_temp_str into the newly allocated buffer. If it did not read a string successfully, it continues with the next iteration of the loop; it does not allocate memory, or change the initial NULL value at the index of the stonesoup_values array.

```
for (stonesoup_i = 0; stonesoup_i < stonesoup_len; ++stonesoup_i) {
    /* STONESOUP: CROSSOVER-POINT (Null Pointer Dereference) */
    if (sscanf(stonesoup_endptr," %79s",stonesoup_temp_str) == 1) {
        stonesoup_values[stonesoup_i] = ((char *)(malloc((strlen(stonesoup_temp_str)
+ 1) * sizeof(char ))));
        strcpy(stonesoup_values[stonesoup_i],stonesoup_temp_str);
        stonesoup_endptr += (strlen(stonesoup_temp_str) + 1) * sizeof(char );
    }
}
```

If the taint source is formatted as expected, when this loop is finished, stonesoup_values has stonesoup_len pointers, each pointing to a valid string on the heap. If the taint source is not formatted as expected, when this loop is finished, stonesoup_values has some pointers that are still NULL.

The snippet then opens a file for writing.

```
stonesoup_temp = fopen(STONESOUP_EXPRESSION_1, "w+");
```

It loops through each element of the stonesoup_values array and outputs them to the file using fputs. If any of the pointers in the stonesoup_values array are still null, this will cause a segmentation fault.

```
for (stonesoup_i = 0; stonesoup_i < stonesoup_len; ++stonesoup_i) {
    fputs(stonesoup_values[stonesoup_i],stonesoup_temp);
    stonesoup_printf(stonesoup_values[stonesoup_i]);
}
```

Benign Inputs

A string that begins with a number and is then followed by that many (or more) space-delimited string.   Other benign inputs are those that do not begin with a number, or begin with a number greater than or equal to 1000. For example:

```
1 foo
```

```
2 a b
```

```
3 alpha bravo charlie
```

Exploiting Inputs

A string that begins with a number between 1 and 999, that is followed by fewer than that many space-delimited string. For example:

```
2 a
```

```
10 a b c
```

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Restrictions and Requirements

C - CWE-476D - NULL Pointer Dereference

Summary

This snippet reads the taint source. If it contains a non-alphanumeric value, the source taint buffer is set to NULL. Subsequently, strcpy is called with the source taint buffer as this source. This causes a null pointer dereference.

In C-CWE476-A, the null pointer dereference arises because of mis-formatting in a comma-separated-values file.

In C-CWE476-B, the null pointer dereference arises because of poor logic in string searching.

In C-CWE476-C, the null pointer dereference arises because of mis-formatting in string parsing.

In C-CWE476-D, the null pointer dereference arises because of unexpected characters in an input string.

In C-CWE476-E, the null pointer dereference arises because of a file open on a file that does not exist.

In C-CWE476-F, the null pointer dereference arises because the function makes a call to a wrapped malloc, but the wrapped malloc has an incorrect implementation and fails to allocate space for a buffer.

In C-CWE476-G, the null pointer dereference arises because the function skipped a necessary call to allocate space for a buffer.

Weakness Class	Number Handling
CWE	CWE-476
Variant	D
Language	C
Status	**WEAK-82** - CWE-476-3-C **DONE**
Negative Technical Impact(s)	DOS: Unconditional Exit

Implementation

The snippet steps through the taint source string, checking that each character is alphanumeric. If it finds a non-alphanumeric character before the terminating null, it breaks out of the loop. When it exits the loop, stonesoup_size is set to the index of the first non-alphanumeric character, or to the index of the terminating null if there are no other non-alphanumeric characters.

```
while(stonesoup_isalnum(STONESOUP_TAINT_SOURCE[stonesoup_size]) && stonesoup_size <
strlen(STONESOUP_TAINT_SOURCE)){
    ++stonesoup_size;
}
```

If the stonesoup_size is not equal to the length of the taint source, the snippet sets the taint source to NULL.

```
/* STONESOUP: CROSSOVER-POINT (Null Pointer Dereference) */
if (stonesoup_size != strlen(STONESOUP_TAINT_SOURCE)) {
    STONESOUP_TAINT_SOURCE = 0;
}
```

It then allocates a buffer of length (stonesoup_size + 1), and copies the taint source into this buffer using strcpy. If the taint source string has been set to NULL, the strcpy causes a segmentation fault.

```
stonesoup_second_buff = malloc((stonesoup_size + 1) * sizeof(char ));
/* STONESOUP: TRIGGER-POINT (Null Pointer Dereference) */
strcpy(stonesoup_second_buff,STONESOUP_TAINT_SOURCE);
```

Benign Inputs

A string with no non-alphanumeric characters. For example:

```
AAAAA
```

```
ABCD123
```

```
1234564760ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
```

Exploiting Inputs

A string with alpha-numeric characters. For example:

```
A.A
```

```
AAAAAAAHHHHHHHHHHHH!
```

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Restrictions and Requirements

C - CWE-476E - NULL Pointer Dereference

Summary

This snippet reads the taint source. If it contains a non-alphanumeric value, the source taint buffer is set to NULL. Subsequently, strcpy is called with the source taint buffer as this source. This causes a null pointer dereference.

In C-CWE476-A, the null pointer dereference arises because of mis-formatting in a comma-separated-values file.

In C-CWE476-B, the null pointer dereference arises because of poor logic in string searching.

In C-CWE476-C, the null pointer dereference arises because of mis-formatting in string parsing.

In C-CWE476-D, the null pointer dereference arises because of unexpected characters in an input string.

In C-CWE476-E, the null pointer dereference arises because of a file open on a file that does not exist.

In C-CWE476-F, the null pointer dereference arises because the function makes a call to a wrapped malloc, but the wrapped malloc has an incorrect implementation and fails to allocate space for a buffer.

In C-CWE476-G, the null pointer dereference arises because the function skipped a necessary call to allocate space for a buffer.

Weakness Class	Number Handling
CWE	CWE-476
Variant	E
Language	C
Status	**WEAK-85** - CWE-476-4-C DONE
Negative Technical Impact(s)	DOS: Unconditional Exit

Implementation

The snippet steps allocates an 80-character buffer on the stack, allocates a pointer to a file.

```
char stonesoup_buffer[80];
FILE *stonesoup_pFile = 0;
```

The snippet then uses the taint source as a file name and opens that file. If the file open fails, then stonesoup_pFile is NULL.

```
/* STONESOUP: CROSSOVER-POINT */
stonesoup_pFile = fopen(STONESOUP_TAINT_SOURCE,"r");
```

It then reads 79 characters from the file it has just opened into stonesoup_buffer. If the file open failed, this causes a read of a NULL pointer and a segmentation fault.

```
/* STONESOUP: TRIGGER-POINT (Null Pointer Dereference: Unchecked file read) */
fgets(stonesoup_buffer,79,stonesoup_pFile);
```

Benign Inputs

The name of an existing file (contents not important). For example:

```
/opt/stonesoup/workspace/testData/good01.txt

This file contains 'This file is not empty.'
```

```
/opt/stonesoup/workspace/testData/good01.txt

This file contains 'Hello world!  And, good-bye.'
```

```
/opt/stonesoup/workspace/testData/good03.txt

This file contains 'Yet another file.'
```

Exploiting Inputs

The name of a file that does not exist. For example:

```
/opt/stonesoup/workspace/testData/bad01.txt

This file does not exist.
```

```
/opt/stonesoup/workspace/testData/bad02.txt

This file does not exist.
```

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment

- Restrictions and Requirements

C - CWE-476F - NULL Pointer Dereference

Summary

This snippet reads the taint source, and converts it to an integer, then an unsigned int. It uses a wrapped malloc to allocate a buffer of the size specified by the taint source. If the size is greater than 512, the wrapped malloc returns NULL. The program attempts to use the buffer, and if the buffer is NULL, this causes a segmentation fault.

In C-CWE476-A, the null pointer dereference arises because of mis-formatting in a comma-separated-values file.

In C-CWE476-B, the null pointer dereference arises because of poor logic in string searching.

In C-CWE476-C, the null pointer dereference arises because of mis-formatting in string parsing.

In C-CWE476-D, the null pointer dereference arises because of unexpected characters in an input string.

In C-CWE476-E, the null pointer dereference arises because of a file open on a file that does not exist.

In C-CWE476-F, the null pointer dereference arises because the function makes a call to a wrapped malloc, but the wrapped malloc has an incorrect implementation and fails to allocate space for a buffer.

In C-CWE476-G, the null pointer dereference arises because the function skipped a necessary call to allocate space for a buffer.

Weakness Class	Number Handling
CWE	CWE-476
Variant	F
Language	C
Status	**WEAK-90** - CWE-476-5-C DONE
Negative Technical Impact(s)	DOS: Unconditional Exit

Implementation

The snippet uses atoi to read an integer from the taint source into stonesoup_buffer_value.

```
stonesoup_buffer_value = atoi(STONESOUP_TAINT_SOURCE);
```

If stonesoup_buffer_value is less than 0, it sets it to 0.

```
if (stonesoup_buffer_value < 0)
    stonesoup_buffer_value = 0;
```

It converts the stonesoup_buffer_value to an unsigned int called stonesoup_size_buffer.

```
stonesoup_size_buffer = ((unsigned int )stonesoup_buffer_value);
```

It then calls my_malloc to allocate a buffer of size stonesoup_size_buffer.

```
stonesoup_malloc_buffer = my_malloc(stonesoup_size_buffer);
```

my_malloc is a function defined in this file. If the input size is 512 or less, it allocates a buffer of that size and returns it. If the input size is greater than 512, it returns NULL pointer.

```
void *my_malloc(unsigned int size)
{
  if (size > 512)
/* STONESOUP: CROSSOVER-POINT */
    return 0;
  return malloc(size);
}
```

The snippet then memsets the allocated buffer to 0. If the buffer was not allocated, a null pointer dereference occurs, causing a segmentation fault.

```
/* STONESOUP: TRIGGER-POINT (Null Pointer Dereference: Wrapped malloc) */
memset(stonesoup_malloc_buffer,0,stonesoup_size_buffer);
```

Benign Inputs

An integer less than or equal to 512. For example:

```
20
```

```
511
```

```
1
```

Exploiting Inputs

An integer greater than 512. For example:

```
1000
```

```
5000
```

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Restrictions and Requirements

C - CWE-476G - NULL Pointer Dereference

Summary

This snippet reads the taint source. If the length of the taint source is 63 bytes or less, it allocates a buffer to copy the taint source into. It then copies the taint source into the buffer, regardless of whether it actually allocated any memory or not. If it did not allocate memory, the buffer pointer is still NULL, and this causes a NULL pointer dereference.

In C-CWE476-A, the null pointer dereference arises because of mis-formatting in a comma-separated-values file.

In C-CWE476-B, the null pointer dereference arises because of poor logic in string searching.

In C-CWE476-C, the null pointer dereference arises because of mis-formatting in string parsing.

In C-CWE476-D, the null pointer dereference arises because of unexpected characters in an input string.

In C-CWE476-E, the null pointer dereference arises because of a file open on a file that does not exist.

In C-CWE476-F, the null pointer dereference arises because the function makes a call to a wrapped malloc, but the wrapped malloc has an incorrect implementation and fails to allocate space for a buffer.

In C-CWE476-G, the null pointer dereference arises because the function skipped a necessary call to allocate space for a buffer.

Weakness Class	Number Handling
CWE	CWE-476
Variant	G
Language	C
Status	**WEAK-93** - CWE-476-6-C DONE
Negative Technical Impact(s)	DOS: Unconditional Exit

Implementation

The snippet allocates a char* on the stack.

```
char *stonesoup_skip_malloc_buffer = 0;
```

If the length of the taint source is 63 bytes or less, the snippet allocates a buffer of size length of taint source + 1.

```
if (strlen(STONESOUP_TAINT_SOURCE) < 63) {
    stonesoup_skip_malloc_buffer = malloc(strlen(STONESOUP_TAINT_SOURCE + 1));
}
```

It then copies the taint source into stonesoup_skip_malloc_buffer, regardless of whether it actually allocated space or not. If it did not allocate space, stonesoup_skip_malloc_buffer is still NULL, and this results in a segmentation fault.

```
/* STONESOUP: TRIGGER-POINT (Null Pointer Dereference: Unchecked strcpy) */
strcpy(stonesoup_skip_malloc_buffer,STONESOUP_TAINT_SOURCE);
```

Benign Inputs

A string with length 63 bytes or less. For example:

```
brah
```

```
dude
```

```
bromigo
```

Exploiting Inputs

A string with length 64 bytes or more. For example:

```
TmFoIG5haCBuYWggbmFoIG5haCBuYWgsIG5haCBuYWggbmFoLCBoZXkgSnVkZQpOYWggbmFoIG5haCBuYWgg
bmFoIG5haCwgbmFoIG5haCBuYWgsIGhleSBKdWRl
```

```
this string is more than sixty four characters long, so it should mess some stuff up
```

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Restrictions and Requirements

Number Handling

This class contains weaknesses related to the handling of numeric values. These weaknesses involve:

- issues that arise from converting from one numeric data type to another
- issues that arise from performing math on numeric data

Weakness Members

The number handling weakness class is defined as a set of weakness types as defined by the MITRE Common Weakness Enumeration (CWE) ontology. The MITRE CWE ontology is defined a a tree, and as such may target CWEs that are a parent or child of another CWE. In those cases, CWEs may be proposed to remove from the test set target given that they are fully covered by one or more other CWEs. Individual weakness variants (i.e. tests) are designed around the definition, description, and code examples provided for each CWE.

CWE	Name	Target Source Languages
CWE-190	Integer Overflow or Wraparound	C, Java
CWE-191	Integer Underflow (Wrap or Wraparound)	C, Java
CWE-194	Unexpected Sign Extension	C, Java
CWE-195	Signed to Unsigned Conversion Error	C, Java
CWE-196	Unsigned to Signed Conversion Error	C, Java
CWE-197	Numeric Truncation Error	C, Java
CWE-369	Divide By Zero	C, Java
CWE-682	Incorrect Calculation	C
CWE-839	Numeric Range Comparison Without Minimum Check	C, Java

Weakness Variants

The following weakness variants are developed and for the C and Java source languages.

C Source Language

Error rendering macro 'detailssummary' : In template Confluence.Templates.Macro.MasterDetail.contentLink2: When evaluating "contextPath()": Error while computing function "contextPath()": null

Java Source Language

Error rendering macro 'detailssummary' : In template Confluence.Templates.Macro.MasterDetail.contentLink2: When evaluating "contextPath()": Error while computing function "contextPath()": null

Runtime Information

Notes

C - CWE-190A - Integer Overflow or Wraparound

Summary

This snippet squares a positive number. If the number is large enough, the square will wrap around and become a negative number.

The snippet then uses the number as a decrementing counter in a while loop. If the number is negative when entering the loop, the loop will never terminate.

This snippet differs from CWE191 because CWE191 is integer underflow and this is integer overflow.

Weakness Class	Number Handling
CWE	CWE-190
Variant	A
Language	C
Status	WEAK-71 - CWE-190-0-C DONE
Negative Technical Impact(s)	DOS: Infinite Loop

Implementation

The snippet takes an input string, and converts it to an integer.

```
stonesoup_tainted_int = atoi(STONESOUP_TAINT_SOURCE);
```

The snippet checks if this integer is greater than 0. If not the snippet exits.

If the integer is greater than 0, the snippet squares the integer.

```
stonesoup_tainted_int = stonesoup_tainted_int * stonesoup_tainted_int;
```

If the integer is large enough, the square will wrap around, and stonesoup_tainted_int will be negative. The snippet has reached the crossover point, where the state of stonesoup_tainted_int has an unintended value.

The snippet then enters a loop, decrementing the stonesoup_tainted_int each time through the loop, and halting when stonesoup_tainted_int equals 0. If stonesoup_tainted_int equals INT_MIN, it is not decremented, so as to prevent an integer underflow.

```
while (stonesoup_tainted_int != 0) {
   /* STONESOUP: TRIGGER-POINT (Integer Overflow) */
            if (stonesoup_tainted_int != INT_MIN) {
                stonesoup_tainted_int--;
            }
   ...
  }
```

The weakness reaches the trigger point in this loop. If stonesoup_tainted_int is positive when the loop begins, the loop exits normally. If stonesoup_tainted_int is negative when the loop begins, the loop will never terminate.

Benign Inputs

Any number less than or equal to the square root of INT_MAX is benign. If INT_MAX is 2147483647, then any number less than or equal to 46340 is benign. For example:

```
50
```

```
100
```

```
10000
```

Due to the fact that none of these values are large enough to cause an integer overflow, they will not result in a bad state.

Exploiting Inputs

Any number greater than the square root of INT_MAX is an exploit. If INT_MAX is 2147483647, then any number greater than 46341 is an exploit. For example:

```
46400
```

```
55000
```

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Restrictions and Requirements

C - CWE-191A - Integer Underflow (Wrap or Wraparound)

Summary

This snippet takes user input string and converts it to an unsigned long long call stonesoup_num. It then caps the value of stonesoup_num at 100. It attempts to malloc (stonesoup_num - 10) pointers. If stonesoup_num is less than 10, stonesoup_num - 10 will underflow, resulting in a request for an enormous amount of memory. Malloc will return NULL, and a subsequent attempt to access the malloc'ed memory will result in a segmentation fault.

This snippet differs from CWE190 because CWE190 is integer overflow and this is integer underflow.

Weakness Class	Number Handling
CWE	CWE-191
Variant	A
Language	C
Status	**WEAK-121** - CWE-191-0-C **DONE**
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

This snippet takes a user input string and converts it to an unsigned long long.

```
unsigned long long stonesoup_num = 0;


stonesoup_num = strtoull(STONESOUP_TAINT_SOURCE, NULL, 10);
```

The snippet checks if stonesoup_num is greater than 0. If it is not, the weakness exits. If it is greater than zero, it checks that the number is no greater than 100, resetting it to 100 if it is.

```
if(stonesoup_num > (unsigned long long) 100 ){
 stonesoup_num = (unsigned long long) 100;
}
```

At this point, 0 <= stonesoup_num <= 100.

The snippet then mallocs room for (stonesoup_num - 10) pointers to characters.

```
stonesoup_buff = malloc((stonesoup_num - (unsigned long long) 10) * sizeof(char *));
```

If stonesoup_num < 10, the subtraction will underflow, and the snippet will attempt to allocate a huge amount of memory. This will result in the memory manager returning a NULL pointer. Subsequently, the snippet will try to write to the pointer, resulting in a segmentation fault.

```
for(stonesoup_i = 0; stonesoup_i < stonesoup_num - (unsigned long long) 10;
stonesoup_i++){
 stonesoup_buff[stonesoup_i] = fgetc(stonesoup_random_data);
}
```

If stonesoup_num = 10, the malloc is for 0 bytes. This is a valid request, and the malloc will return a valid pointer. However, the space allocated is 0 bytes long. Writing to this space is invalid, and unintended behavior. However, segmentation fault does not reliably. Furthermore, no integer underflow has occurred. Accordingly, T&E does not use a string of 10 for any of its inputs.

Benign Inputs

All inputs greater than 10 are benign.

```
45
```

```
80
```

```
8000
```

Exploiting Inputs

All inputs between 1 and 9 are exploit inputs. The input value 10 is an invalid value, but does not cause and integer underflow or a segmentation fault.

```
8
```

```
4
```

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Restrictions and Requirements

C - CWE-191B - Integer Underflow (Wrap or Wraparound)

Summary

This snippet takes a filename. It determines the size of the file, and attempts to check whether the size of the file is smaller than 128 characters. The arithmetic used in the if-check may cause an integer underflow, which will result in the if-check succeeding when it should fail. If the if-check succeeds after an integer underflow, a large amount of data is copied into a small (128-character) buffer on the stack. This will overwrite memory on the stack, resulting in segmentation fault upon return from the function.

This snippet differs from CWE190 because CWE190 is integer overflow and this is integer underflow.

Weakness Class	Number Handling
CWE	CWE-191
Variant	B
Language	C
Status	**WEAK-408** - CWE-191-1-C DONE
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

This snippet creates a buffer on the stack with room for 128 characters. It creates two shorts: stonesoup_max_size and stonesoup_remaining_space. It creates an int called stonesoup_size, and a file handle called stonesoup_file.

```
char stonesoup_buff[128];
short stonesoup_max_size = 128;
short stonesoup_remaining_space = 0;
int stonesoup_size = 0;
FILE * stonesoup_file = 0;
```

The snippet takes a user input string and attempts to open a file by that name for reading.

```
stonesoup_file = fopen(STONESOUP_TAINT_SOURCE, "r");
```

If the file exists, the snippet seeks to the end of the file, puts the size of the file into the int stonesoup_size, then seeks back to the beginning of the file.

```
fseek(stonesoup_file, 0, SEEK_END);
stonesoup_size = ftell(stonesoup_file);
fseek(stonesoup_file, 0, SEEK_SET);
```

The snippet aims to copy data from the file into stonesoup_buff. In attempt to not overflow the buffer, it calculates how much space would be left over after the file contents are copied. It calculates stonesoup_remaining_space by subtracting stonesoup_isize from stonesoup_max_size. If stonesoup_size is large enough, this will underflow, resulting in a large positive number. This is the cross-over point.

```
/* STONESOUP: CROSSOVER-POINT (Integer Underflow) */
stonesoup_remaining_space = stonesoup_max_size - stonesoup_size;
```

The snippet checks that there the copy into stonesoup_buff is allowed by checking if stonesoup_remaining_space is greater than 0. If an underflow has happened, the if-check will pass, but the copy will overwrite the available space in stonesoup_buff.

This will overwrite the stack, eventually corrupting the return pointer. A segmentation fault will occur upon return from the current function.

Benign Inputs

A filename that does not exist is benign.

A filename pointing to a file that has 127 or fewer character is benign, because it can be copied safely into the buffer.

A filename pointing to a file that contains more than 127 characters and fewer than SHRT_MAX + 128 + 2 = 32897 characters is benign, because no underflow happens. The if-check works as intended and prevents the buffer copy from happening.

```
input = "/opt/stonesoup/workspace/testData/good_1.txt"

/opt/stonesoup/workspace/testData/good_1.txt contains:
This_is_a_file
```

```
input = "/opt/stonesoup/workspace/testData/good_2.txt"

/opt/stonesoup/workspace/testData/good_2.txt contains:
This_is_a_file_that_has_more_data
```

```
input = "/opt/stonesoup/workspace/testData/good_3.txt"

/opt/stonesoup/workspace/testData/good_3.txt contains:
Some_sort_of_file_with_data_in_it.
```

Exploiting Inputs

A filename pointing to a file that contains greater than or equal to than SHRT_MAX + 128 + 2 = 32897 is an exploit, because an underflow happens. The if-check does not work as intended, and a large amount of data is copied into a small buffer. This corrupts the stack, resulting in segmentation fault upon return from the function.

```
input = "/opt/stonesoup/workspace/testData/bad_1.txt"

/opt/stonesoup/workspace/testData/bad_1.txt contains 40,000 copies of "a"
```

```
input = "/opt/stonesoup/workspace/testData/bad_1.txt"

/opt/stonesoup/workspace/testData/bad_1.txt contains 45,000 copies of "b"
```

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Restrictions and Requirements

C - CWE-194A - Unexpected Sign Extention

Summary

This snippet converts a user string to a short, and then converts that short to an unsigned int. If the short is negative, this will result in unexpected sign extension. The unsigned int value is used to determine how much data to read from a file, resulting in massive buffer overwrite if the file is large and the original input was negative.

Weakness Class	Number Handling
CWE	CWE-194
Variant	A
Language	C
Status	**WEAK-125** - CWE-194-0-C DONE

Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

This snippet takes a string allocates an unsigned int called stonesoup_to_unsign, and a buffer called stonesoup_buff.

```
unsigned int stonesoup_to_unsign = 0;
char *stonesoup_buff = 0;
```

The snippet allocates a 30,000-character buffer and assigns it to stonesoup_buff.

```
stonesoup_buff = ((char *)(malloc(30000  * sizeof(char ))));
```

The snippet passes the (possibly tainted) user input to a function called stonesoup_get_int_value, which converts a string to a short. This function limits the return value to a number between -30000 and 30000.

```
short stonesoup_get_int_value(char *ss_tainted_buff)
{
  short to_short = 0;
  int tainted_int = 0;
  tainted_int = atoi(ss_tainted_buff);
  if (tainted_int != 0) {
    if (tainted_int > 30000)
      tainted_int = 30000;
    if (tainted_int < -30000)
      tainted_int = -30000;
    to_short = ((short )tainted_int);
  }
  return to_short;
}
```

The value returned from stonesoup_get_int_value is silently cast to an unsigned int in the main weakness function:

```
/* STONESOUP: CROSSOVER-POINT (Unexpected Sign Extension) */
stonesoup_to_unsign = stonesoup_get_int_value(STONESOUP_TAINT_SOURCE);
```

This is the crossover point. If the original user value is negative, stonesoup_to_unsign will contain an extremely large positive number, as a result of unexpected sign extension.

Subsequently, the program opens a file for reading, with the filename specified by the user:

```
stonesoup_file = fopen(STONESOUP_EXPRESSION_1,"r");
```

The code repeatedly reads 1,000-character chunks into the 30,000-character buffer, stonesoup_buff.

```
while (((unsigned int )stonesoup_counter) < stonesoup_to_unsign) {
            /* STONESOUP: TRIGGER-POINT (Unexpected Sign Extension) */
            stonesoup_bytes_read = fread(&stonesoup_buff[stonesoup_counter],
                sizeof(char), 1000, stonesoup_file);
            if (stonesoup_bytes_read == 0) {
                break;
            }
            stonesoup_counter += stonesoup_bytes_read;
        }
```

If stonesoup_to_unsign has undergone unexpected sign extension, and there is enough data in the file specified by STONESOUP_EXPRESSION_1, this may result in reading far more than 30,000 bytes, and will eventually cause a segmentation fault.

Benign Inputs

Benign inputs are non-negative values. For example:

```
50
# Requires running the following pre-process to initialize myfile.txt:
dd if=/dev/urandom of=/opt/stonesoup/workspace/testData/myfile.txt bs=1 count=1M
```

```
1000
# Requires running the following pre-process to initialize myfile.txt:
dd if=/dev/urandom of=/opt/stonesoup/workspace/testData/myfile.txt bs=1 count=1M
```

```
20000
# Requires running the following pre-process to initialize myfile.txt:
dd if=/dev/urandom of=/opt/stonesoup/workspace/testData/myfile.txt bs=1 count=1M
```

Exploiting Inputs

Exploiting inputs are all negative values. For example:

```
-5
# Requires running the following pre-process to initialize myfile.txt:
dd if=/dev/urandom of=/opt/stonesoup/workspace/testData/myfile.txt bs=1 count=1M
```

```
-10
# Requires running the following pre-process to initialize myfile.txt:
dd if=/dev/urandom of=/opt/stonesoup/workspace/testData/myfile.txt bs=1 count=1M
```

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Restrictions and Requirements

C - CWE-195A - Signed to Unsigned Conversion Error

Summary

This snippet allocates a stack buffer of size 16. It takes the user input and performs a check to see if it is possible to copy the user input into a 16-byte buffer. If the user input is larger than 15 bytes, then the check method returns a -1. This value is converted to an unsigned type (size_t), resulting in an enormous number if the check method returned -1. The snippet then uses that unsigned value as a size for the copy from the user input into the stack buffer. Because the unsigned value is huge, this overwrites the stack, resulting in a segmentation fault upon return from the function.

This weakness differs from CWE196 because CWE195 is unsigned to signed conversion error and this is signed to unsigned conversion error.

Weakness Class	Number Handling
CWE	CWE-195
Variant	A
Language	C
Status	**WEAK-128** - CWE-195-0-C **DONE**
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

This snippet declares a buffer called stonesoup_dest of size 16 on the stack. It declares a size_t (which is an unsigned type) called stonesoup_size_var.

```
const int stonesoup_MAXLEN = 16;
    char stonesoup_dest[stonesoup_MAXLEN];
    size_t stonesoup_size_var = 0;
```

The snippet sets the stonesoup_dest buffer to have all x's, with a final terminating '\0' character at index 15.

```
memset(stonesoup_dest,'x',stonesoup_MAXLEN);
    stonesoup_dest[stonesoup_MAXLEN - 1] = '\0';
```

The main weakness function makes a call to stonesoup_returnChunkSize, and assigns the result to the unsigned variable stonesoup_size_var.

```
stonesoup_size_var = stonesoup_returnChunkSize(stonesoup_dest,
STONESOUP_TAINT_SOURCE);
```

The function stonesoup_returnChunkSize compares the length of the two strings passed to it. If the destination string (the first parameter) is smaller than the source string (the second parameter), the function returns -1 to indicate that the copy is not safe. If the destination string is

larger than the source string, the function returns a non-negative number, which is the length of the destination string.

```
int stonesoup_returnChunkSize(void *dest,void *src)
{
  if (strlen(dest) < strlen(src)) {
/* STONESOUP: CROSSOVER-POINT (Signed To Unsigned Conversion Error) */
    return -1;
  }
  return strlen(dest);
}
```

The main weakness function takes the int returned by stonesoup_returnChunkSize and puts it into an unsigned variable. If the return value is negative, you will get a signed to unsigned conversion error, and stonesoup_size_var wil contain a very large number.

The main weakness function then checks that stonesoup_size_var is greater than 0. If it is, it copy stonesoup_size_var characters from the user string to the destination string. If the signed to unsigned conversion error has occurred, the copy will be very large, resulting in an overread of STONESOUP_TAINT_SOURCE, and an overwrite of stonesoup_dest.

```
if (stonesoup_size_var > 0)
        memcpy(stonesoup_dest, STONESOUP_TAINT_SOURCE, stonesoup_size_var);
```

The overwrite will corrupt the stack, eventually over-writing the return pointer. A segmentation fault will occur when the main weakness function returns.

Benign Inputs

Benign inputs are string values with length less than 15.

```
Good_Bye_World
```

```
abcd
```

```
short stmt
```

Exploiting Inputs

Exploiting inputs are string values strings with length 16 or more characters.

```
Heeellllloooooo_Wooooorrrrrllllldddddd!!!!!!
```

```
this_string_is_greater_than_16_characters
```

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Restrictions and Requirements

C - CWE-196A - Unsigned to Signed Conversion Error

Summary

This weakness reads a number to be used as a loop counter. The loop counter is initially read as an unsigned long, then converted to an int. If the number read in is larger than MAX_UINT, it is silently converted to a negative number. This breaks the loop counter logic, resulting in an infinite loop.

This weakness differs from CWE195 because CWE195 is signed to unsigned conversion error and this weakness is unsigned to signed conversion error.

Weakness Class	Number Handling
CWE	CWE-196
Variant	A
Language	C
Status	WEAK-131 - CWE-196-0-C DONE
Negative Technical Impact(s)	DOS: Infinite Loop

Implementation

The snippet allocates a signed int on the stack called stonesoup_size:

```
int stonesoup_size = 0;
```

The snippet calls the function stonesoup_get_size to determine the size of the user input string:

```
stonesoup_num = stonesoup_get_size(STONESOUP_TAINT_SOURCE);
```

The function stonesoup_get_size puts the length of the input string into an unsigned long. It then returns that unsigned long as an unsigned int:

```
unsigned int stonesoup_get_size(char *ss_tainted)
{
  unsigned long uns_int = 0UL;
  uns_int = strtoul(ss_tainted,0,0);
  if (uns_int > ((unsigned long )4294967295U) ||
      uns_int == 0)
    uns_int = 1U;
  return (unsigned int )uns_int;
}
```

If the original input is between 0 and 127 inclusive, the cast will not result in a negative number, and the snippet will complete without error.

Benign Inputs

All inputs between 0 and 127 inclusive are benign. For instance:

```
2
```

```
8
```

Since both of these inputs are in the range stated above they will not cause an unsigned to signed conversion error, and are thus benign.

Exploiting Inputs

All inputs between 128 and 255 inclusive are benign. For instance:

```
-1
```

```
-10
```

Since both of these inputs are in the range stated above they will cause an unsigned to signed conversion error, and are thus exploitative.

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Restrictions and Requirements

C - CWE-197A - Numeric Truncation Error

Summary

This snippet takes an unsigned long value and uses it in an initialization function for a struct. Within the initialization function, the long gets converted to an unsigned short when a struct uses the unsigned long as an initialization value for an unsigned short member. If the unsigned

long number is large enough, it will result in a numerical truncation.

The snippet then uses the short value to malloc memory for a char * buffer. It then memset's the char * member of the struct to ascii value 98 ('b') for the size of the unsigned long value. If the numerical truncation took place, then the char * buffer won't have enough memory allocated to it for this to properly work. Which will result in writing 'garbage' data into memory somewhere, potentially corrupting the program.

Weakness Class	Number Handling
CWE	CWE-197
Variant	A
Language	C
Status	**WEAK-135** - CWE-197-0-C **DONE**
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

The snippet takes an input string, and converts it to an unsigned long.

```
stonesoup_number = strtol(STONESOUP_TAINT_SOURCE,0U,10);
```

The snippet checks if this unsigned long is greater than 0. If not the snippet exits.

If the unsigned long is greater than 0, the snippet sends the integer to an initialization function.

```
struct stonesoup_struct_data {
  char *buffer_member;
  unsigned short size_member;
};

struct stonesoup_struct_data *stonesoup_data = 0;

stonesoup_data = stonesoup_init_data(stonesoup_number);
```

If the unsigned long is large enough, the initialization function, will set an unsigned short in a struct that it is initializing to a value that is larger than it can hold. At this point the snippet has reached the crossover point, where the state of the unsigned short has an unintended value due to numerical truncation. The initialization function then uses the short value to malloc memory for the buffer within the struct it is initializing. It then correctly fills that buffer using the short value, whether it is truncated or not.

```
struct stonesoup_struct_data *stonesoup_init_data(long number_param)
{
  struct stonesoup_struct_data *init_data_ptr = 0;
  init_data_ptr = ((struct stonesoup_struct_data *)(malloc(sizeof(struct
stonesoup_struct_data ))));
  if (init_data_ptr == 0)
    return 0;
  init_data_ptr -> size_member = 0;
/* STONESOUP: CROSSOVER-POINT (Numerical Truncation Error) */
  init_data_ptr -> size_member = number_param;
  init_data_ptr -> buffer_member = ((char *)(malloc(sizeof(char ) * init_data_ptr ->
size_member)));
  if (init_data_ptr -> buffer_member == 0) {
    free(init_data_ptr);
    return 0;
  }
  memset(init_data_ptr -> buffer_member,'a',init_data_ptr -> size_member);
  init_data_ptr -> buffer_member[init_data_ptr -> size_member - 1] = 0;
  return init_data_ptr;
}
```

After the initialization function returns an initialized struct, the main function then attempts to fill the structs buffer with 'a's. However it uses the unsigned long from the source taint instead of the unsigned short from the struct. If the source taint was a large enough number, then this will result in an overflow of data into memory and crash the program.

```
/* STONESOUP: TRIGGER-POINT (Numerical Truncation Error) */
memset(stonesoup_data -> buffer_member, 98, stonesoup_number);
stonesoup_data -> buffer_member[stonesoup_number - 1] = 0;
stonesoup_printf("%s\n", stonesoup_data -> buffer_member);
```

Benign Inputs

Any number less than USHRT_MAX: 65535 is benign. For example:

```
1
```

```
10
```

```
1000
```

Due to the fact that none of these values are large enough to cause a Numerical Truncation Error, they will not result in a bad state.

Exploiting Inputs

Any number greater than USHRT_MAX: 65535 is an exploit. For example:

```
65538
```

```
131074
```

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Restrictions and Requirements

C - CWE-369A - Divide By Zero

Summary

This snippet takes an integer and mods it by four. That resulting number is then used to divide the number 1024, and the result is then printed. If the source integer is directly divisible by 4, this will result in a divide by zero error.

Weakness Class	Number Handling
CWE	CWE-369
Variant	A
Language	C
Status	**WEAK-140** - CWE-369-0-C **DONE**
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

This snippet take a string as input and converts it to an integer.

```
stonesoup_input = atoi(STONESOUP_TAINT_SOURCE);
```

If the input isn't zero, it then mods the input by 4 resulting in the cross-over point. It then uses the result to divide 1024 and print the result. If the source input is zero, then nothing happens.

```
if (stonesoup_input != 0) {
        /* STONESOUP: CROSSOVER-POINT (Divide By Zero) */
        stonesoup_mod = stonesoup_input % 4;
        /* STONESOUP: TRIGGER-POINT (Divide By Zero) */
        stonesoup_quotient = 1024 / stonesoup_mod;
        stonesoup_printf("%f\n", stonesoup_quotient);
}
```

Benign Inputs

Benign inputs consist of integer numbers not divisible by four.

Examples are as follows:

```
99
```

```
-2
```

```
1023
```

Exploiting Inputs

Exploiting inputs consist of integer numbers divisible by four

Examples are as follows:

```
1024
```

```
-512
```

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Restrictions and Requirements

C - CWE-682A - Incorrect Calculation

Summary

This weakness reads a number and attempts to modify the two high bytes of it, if it is greater than 65535. If the calculation to modify the two high bytes is performed, it will happen incorrectly due to the placement of the pointer modification and it instead changes the bytes on function pointers that were created instead.. When these function pointers are then called, it results in an error. If the calculation doesn't take place, then the function calls will execute without a problem.

Weakness Class	Number Handling
CWE	CWE-682
Variant	A
Language	C
Status	WEAK-144 - CWE-682-0-C DONE
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

The snippet creates several function pointers that surround an unsigned long value as well as some byte values and an unsigned long pointer:

```
void (*stonesoup_function_ptr_1)() = 0;
void (*stonesoup_function_ptr_2)() = 0;
unsigned long stonesoup_input_num;
void (*stonesoup_function_ptr_3)() = 0;
void (*stonesoup_function_ptr_4)() = 0;
char *stonesoup_byte_4 = 0;
char *stonesoup_byte_3 = 0;
unsigned long *stonesoup_ptr = 0;
```

The snippet then checks that the input is a valid and assigns its unsigned long value and sets a pointer to that value as well:

```
if (strlen(STONESOUP_TAINT_SOURCE) >= 1 && STONESOUP_TAINT_SOURCE[0] != '-') {
     stonesoup_input_num = strtoul(STONESOUP_TAINT_SOURCE,0U,16);
     stonesoup_ptr = &stonesoup_input_num;
}
```

The unsigned long pointer value is then checked to see if it is greater than 65535. If it is, then the incorrect calculation is performed on the two high bytes of the pointer:

```
if ( *stonesoup_ptr > 65535) {
     /* STONESOUP: CROSSOVER-POINT (Incorrect Calculation) */
     stonesoup_byte_3 = ((char *)(stonesoup_ptr + 2));
     stonesoup_byte_4 = ((char *)(stonesoup_ptr + 3));
     *stonesoup_byte_3 = 0;
     *stonesoup_byte_4 = 0;
}
```

Finally the function pointers are called. If the incorrect calculation took place, then one or more of them will have been modified, and therefore incorrect. When called this will result in the program failing.

```
/* STONESOUP: TRIGGER-POINT (Incorrect Calculation) */
stonesoup_function_ptr_1();
stonesoup_function_ptr_2();
stonesoup_function_ptr_3();
stonesoup_function_ptr_4();
```

Benign Inputs

All inputs between 0 and 65535 inclusive in base 16 are benign. For instance:

```
10
```

```
0
```

```
FFFF
```

Exploiting Inputs

All inputs greater than 65535 in base 16. For Instance:

```
10000
```

```
FFFFFF
```

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Restrictions and Requirements

C - CWE-682B - Incorrect Calculation

Summary

This weakness reads a number and attempts to modify the two high bytes of it, if it is greater than 65535. If the calculation to modify the two high bytes is performed, it will happen incorrectly due to the placement of the pointer modification and it instead changes the bytes on function pointers within a struct that were created instead. When these function pointers are then called, it results in an error. If the calculation doesn't take place, then the function calls will execute without a problem.

Weakness Class	Number Handling
CWE	CWE-682
Variant	B
Language	C
Status	**WEAK-144** - CWE-682-0-C **DONE**
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

The snippet creates an unsigned int pointer and a struct which gets function pointers initialized:

```
struct stonesoup_struct {
    void (*stonesoup_function_ptr_1)();
    unsigned int stonesoup_input_num;
    void (*stonesoup_function_ptr_2)();
};


char *stonesoup_byte_4 = 0;
char *stonesoup_byte_3 = 0;
unsigned int *stonesoup_ptr = 0;
struct stonesoup_struct ssS;
ssS.stonesoup_function_ptr_1 = stonesoup_function;
ssS.stonesoup_function_ptr_2 = stonesoup_function;
```

The snippet then checks that the input is a valid and assigns its unsigned long to the unsigned int in the structvalue and sets a pointer to that value as well:

```
if (strlen(STONESOUP_TAINT_SOURCE) >= 1 && STONESOUP_TAINT_SOURCE[0] != '-') {
     ssS.stonesoup_input_num = strtoul(STONESOUP_TAINT_SOURCE,0U,16);
     stonesoup_ptr = &(ssS.stonesoup_input_num);
}
```

The unsigned int pointer value is then checked to see if it is greater than 65535. If it is, then the incorrect calculation is performed on the two high bytes of the pointer:

```
if ( *stonesoup_ptr > 65535) {
     /* STONESOUP: CROSSOVER-POINT (Incorrect Calculation) */
     stonesoup_byte_3 = ((char *)(stonesoup_ptr + 2));
     stonesoup_byte_4 = ((char *)(stonesoup_ptr + 3));
     *stonesoup_byte_3 = 0;
     *stonesoup_byte_4 = 0;
}
```

Finally the function pointers are called. If the incorrect calculation took place, then one of them will have been modified, and therefore incorrect. When called this will result in the program failing.

```
/* STONESOUP: TRIGGER-POINT (Incorrect Calculation) */
ssS.stonesoup_function_ptr_2();
```

Benign Inputs

All inputs between 0 and 65535 inclusive in base 16 are benign. For instance:

```
10
```

```
0
```

```
FFFF
```

Exploiting Inputs

All inputs greater than 65535 in base 16. For Instance:

```
10000
```

```
FFFFFF
```

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment

- Restrictions and Requirements

C - CWE-839A - Divide By Zero

Summary

This snippet takes an integer and checks for an upper limit. If the number is less than the upper limit, then, a buffer the size of the upper limit will be created and filled with 'a's. The buffer will then be filled with 'b's using the input integer. If the input is negative, this will result in an underwrite.

Weakness Class	Number Handling
CWE	CWE-839
Variant	A
Language	C
Status	WEAK-147 - CWE-839-0-C DONE
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

This snippet take a string as input and converts it to an integer.

```
int stonesoup_len;
stonesoup_len = atoi(STONESOUP_TAINT_SOURCE);
```

If the input is less than the upper limit of 4096, then a buffer will be created and filled with 'a's.

```
/* STONESOUP: CROSSOVER-POINT (No Minimum Check) */
    if (stonesoup_len < 4096) {
        stonesoup_buffer = ((char *)(malloc(4096 * sizeof(char ))));
        if (stonesoup_buffer != 0) {
            memset(stonesoup_buffer,'a',4096);
```

From there, the buffer will then get some of the 'a's replaces with 'b's using the input number. If the input number is negative, this results in an underwrite to the negative starting position in the buffer arrary, to the end of the array.

```
/* STONESOUP: TRIGGER-POINT (No Minimum Check) */
            memset(&stonesoup_buffer[stonesoup_len],'b',4096 - stonesoup_len);
            stonesoup_buffer[4095] = 0;
```

Benign Inputs

Benign inputs consist of integer numbers that are positive.

Examples are as follows:

```
10
```

```
1000
```

```
20
```

Exploiting Inputs

Exploiting inputs consist of integer numbers that are negative

Examples are as follows:

```
-8000000
```

```
-16777215
```

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Restrictions and Requirements

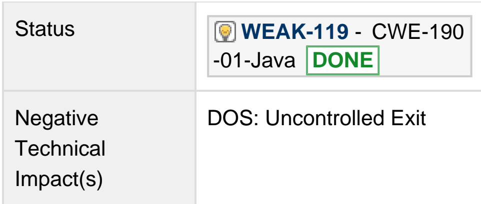
J - CWE-190A - Integer Overflow or Wraparound

Summary

This snippet implements an unchecked addition on an input number. This input is taken as a short, and is added with a number such that if the input is greater than or equal to 500, the resulting value overflows the type short and wraps around to negative before being used to allocate an array, causing a NegativeArraySizeException to be thrown. If the input number is less than 500, no overflow occurs and the allocation, and remainder of the snippet, execute without issue.

This snippet differs from CWE191 because CWE191 is integer underflow and this is integer overflow.

Weakness Class	Number Handling
CWE	CWE-190
Variant	A
Language	Java



Status	WEAK-119 - CWE-190-01-Java DONE
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

The snippet takes an input number as a short and verifies that it is non-negative, setting it to 0 if it is.

```
if (stonesoup_checked_value < 0) {
 stonesoup_checked_value = 0;
}
```

The snippet then adds this to Short.MAX_VALUE - 500, saving the value as a short. The weakness has now reached the crossover point, or the point at which the integer overflow has occurred, leaving the program in a bad state.

```
/* STONESOUP: crossover point (user controlled value) */
short stonesoup_value = (short) (((short)(Short.MAX_VALUE) -
                (short)500) + stonesoup_checked_value);
```

After this, the snippet attempts to allocate an array of size stonesoup_value. The weakness has now reached the trigger point. If stonesoup_value has overflowed, the array allocation fails with a NegativeArraySizeException. If stonesoup_value has not overflowed, the array is allocated successfully and the snippet completes without error.

Benign Inputs

Any number less than 500 is benign. For example:

```
200
```

```
400
```

Due to the fact that none of these values are large enough to cause an integer overflow, they will not result in a bad state.

Exploiting Inputs

Any number greater than or equal to 500. For example:

```
500
```

```
600
```

Since both of these values are larger than or equal to 500, they will cause an integer overflow, resulting in a bad state.

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Restrictions and Requirements

J - CWE-190B - Integer Overflow or Wraparound

Summary

This snippet takes user input in the form of a number (type short, -32768 to 32767), implements an insufficient check, and uses this input as the incrementing value for a loop counter. If this user controlled value is sufficiently large the calculation will cause the loop counter to overflow to a negative number, resulting in an infinite loop.

This snippet differs from CWE191 because CWE191 is integer underflow and this is integer overflow.

Weakness Class	Number Handling
CWE	CWE-190
Variant	B
Language	Java
Status	**WEAK-119** - CWE-190-01-Java DONE
Negative Technical Impact(s)	DOS: Infinite Loop

Implementation

The snippet takes in a number and checks that the input is > 0, setting it to 1 if it not.

```
if (stonesoup_checked_value <= 0) {
    stonesoup_checked_value = 1;
    output.println("resetting value to 1");
}
```

The snippet then enters a while loop which checks whether the loop counter (stonesoup_counter, initialized to zero) is less than 10. The entrance to this loop is the crossover point. If the loop counter is less than 10, it is incremented by adding the user controlled input to it. This is now the trigger point. If this user controlled input is either 32767 or 32766, the loop counter will overflow to negative. Since the loop counter is only incremented when it is greater than 0, this results in an infinite loop.

```
/* STONESOUP: crossover point (user controlled value) */
while(stonesoup_counter < 10){
 output.println("Loop #" + stonesoup_counter);
 if (stonesoup_counter > 0){
  Tracer.tracepointMessage("TRIGGER-POINT: BEFORE");
  /* STONESOUP: trigger point (overflows, causing infinite loop) */
  stonesoup_counter += stonesoup_checked_value;
 }
 if (stonesoup_counter > 0 || ++lttngCtr >= 100) {
  lttngCtr = 1;
  Tracer.tracepointVariableShort("stonesoup_counter", stonesoup_counter);
 }
}
```

If the user controlled input was not an exploiting value, the snippet will exit the loop and complete normally.

Benign Inputs

Benign inputs are all inputs between -32768 and 32765 inclusive. For example:

```
2000
```

```
4000
```

Since neither of these inputs are outside of the given range, they are benign inputs.

Exploiting Inputs

Exploiting inputs are the two inputs that will cause integer overflow when added to a short of value of 1. These two inputs are as follows:

```
32766
```

```
32767
```

Both of these inputs will overflow the loop counter, causing the exploit.

Source Code

For more information, please refer to the weakness Source Code.

Related Information

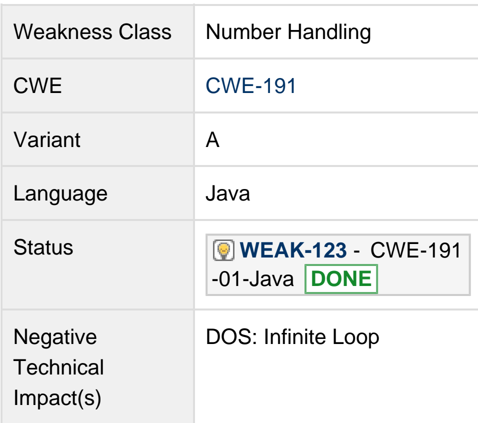
- Execution Environment
- Restrictions and Requirements

J - CWE-191A - Integer Underflow (Wrap or Wraparound)

Summary

This snippet takes user input in the form of a number (type short, -32768 to 32767), implements an insufficient check, and uses this value as the decrement for a loop counter. If this user controlled value is sufficiently large, it will cause the calculated loop counter to underflow to a positive value resulting in an infinite loop.

This snippet differs from CWE190 because CWE190 is integer overflow and this is integer underflow.



Weakness Class	Number Handling
CWE	CWE-191
Variant	A
Language	Java
Status	WEAK-123 - CWE-191 -01-Java DONE
Negative Technical Impact(s)	DOS: Infinite Loop

Implementation

This snippet takes user input in the form of a number, type short, and checks to make sure the number is grater than or equal to 0.

```
if (stonesoup_checked_value < 0) {
 stonesoup_checked_value = 0;
}
```

An array of data is allocated along with a counter and offset to use while iterating through this newly allocated array of data.

```
Short[] stonesoup_some_values = new Short[] { 0, 1, 2, 3, 4, 5, 6, 7,
    8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20 };
short stonesoup_counter = -20;
short stonesoup_offset = 40;
```

The snippet then enters the crossover point, looping through the array of data, and subtracting the user controlled input from the array counter, resetting the counter to -20 if it is ever greater than -20. If the user controlled input is sufficiently large, this subtraction will result in integer overflow, causing the counter to be a large positive number, and subsequently get reset to -20 resulting in an infinite loop. This subtraction and resetting is the trigger point.

```
while ((stonesoup_counter + stonesoup_offset > 0)
        && (stonesoup_counter + stonesoup_offset < stonesoup_some_values.length)) {
    output.printf("stonesoup_some_values[%d] : %s\n", stonesoup_counter
            + stonesoup_offset, stonesoup_some_values[stonesoup_counter
            + stonesoup_offset]);
 {...}
    /* STONESOUP: trigger point (underflows, causing infinite loop) */
    stonesoup_counter -= stonesoup_checked_value;
    if (stonesoup_counter > -20) {
        stonesoup_counter = -20;
    }
 {...}
}
```

If the user controlled input was not sufficiently large, the loop will complete successfully, causing the rest of the snippet to run without error.

Benign Inputs

All inputs between -32768 and 32748, inclusive are benign. For example:

```
1
```

```
10
```

Since both of these inputs are within the range stated above they do not cause integer underflow and are thus benign.

Exploiting Inputs

All inputs between 32749 and 32767, inclusive, are exploiting inputs. For example:

```
32767
```

```
32765
```

Since both of these inputs are withing the range stated above they cause integer underflow and are thus exploitative.

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Restrictions and Requirements

J - CWE-194A - Unexpected Sign Extention

Summary

This snippet implements two casts that can result in unexpected sign extension. The snippet takes input in the form of a user controlled number (type short) in the range of 1 to 255. This input is used to allocate an array, then cast to a byte and re-cast to a char. This casting can lead to an unexpected sign extension and result in an exception when used to iterate though the previously allocated array.

Weakness Class	Number Handling
CWE	CWE-194
Variant	A
Language	Java
Status	WEAK-127 - CWE-194 -01-Java DONE
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

This snippet takes user input in the form of a number, type short, in the range 1 to 255, inclusive. This input is then check to make sure it is in that range, and set to 1 or 255 if it is either below or above the range respectively.

```
if (stonesoup_array_size < 0) {
 stonesoup_array_size = 0;
} else if (stonesoup_array_size > 255) {
    stonesoup_array_size = 255;
}
```

Once the data is (partially) sanitized it is cast to a signed byte and stored. This cast will cause the byte (range -128 to 127) to be interpreted as a negative number if the original is between 128 and 255, inclusive. An array is then allocated with size equal to the original sanitized input (stonesoup_array_size).

```
byte stonesoup_counter_max_signed = (byte)stonesoup_array_size;
Tracer.tracepointVariableByte("stonesoup_counter_max_signed",
stonesoup_counter_max_signed);

/* array to initialize */
int[] stonesoup_array = new int[stonesoup_array_size];
```

The byte representation of the input number is then cast to a char (unsigned), possibly causing unintended sign extension through the promotion to unsigned. This is the crossover point.

```
/* STONESOUP: crossover point (promotion to unsigned causes unexpected sign
extension) */
char stonesoup_counter_max = (char) stonesoup_counter_max_signed;
```

The snippet then attempts to initialize the previously allocated array to all 1s by iterating through the array from position 0 to stonesoup_counter_max, the newly cast char. If the origional input was between 128 and 255 inclusive, this char will have experienced sign extension during the cast which will cause it to be interpreted as a large positive number, resulting in an ArrayIndexOutOfBounds exception.

```
for (char counter = 0; counter < stonesoup_counter_max; counter++) {
 /* STONESOUP: trigger point (sign extension causes out of bounds exception) */
 stonesoup_array[counter] = 1;
}
```

If the original input was between 1 and 127 inclusive the unintended sign extension would not have taken place, the array would be initialized properly, and the snippet would execute to completion without error.

Benign Inputs

Benign inputs are all inputs from -32768 to 127 inclusive, sanitized to 1 to 127 by the snippet. For example:

```
1
```

```
20
```

Since both of these values are within the range above they do not cause unintended sign extension and are thus benign.

Exploiting Inputs

Exploiting inputs are all inputs from 128 to 32767 inclusive, sanitized to 128 to 255 by the snippet. For example:

```
128
```

```
255
```

Since both of these values are within the range above they cause unintended sign extension and are thus exploitative.

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Restrictions and Requirements

J - CWE-195A - Signed to Unsigned Conversion Error

Summary

This snippet takes user input in the form of an number (type short, range -32768 to 32767 inclusive), allocates an array with size equal to the absolute value of the input number, casts the input to an unsigned char, and uses this unsigned char to index the array. If the original input is a negative number, the cast to an unsigned char will cause the value to be interpreted as a large positive value causing an ArrayIndexOutOfBounds exception.

This weakness differs from CWE196 because CWE195 is unsigned to signed conversion error and this is signed to unsigned conversion error.

Weakness Class	Number Handling
CWE	CWE-195
Variant	A
Language	Java
Status	WEAK-130 - CWE-195 -01-Java DONE
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

This snippet takes user input in the form of a number (type short, range -32768 to 32767) and allocates an array with size equal to the absolute value of the user input. The user input is then cast to a char, and since the types short and char are the same size in memory, the cast from short to char preserves the binary representation of the data, causing a negative short to be interpreted as a large positive char (the highest order bit is interpreted as sign in a signed short, and as a numeric value in an unsigned char). This is now the crossover point.

```
/* STONESOUP: crossover point (user controlled value, convert from signed to
unsigned) */
int[] stonesoup_array = new int[Math.abs(value)];
char stonesoup_max_char = (char)((short)value);
```

This char representation of the user input is then used to index the previously allocated array, stepping through each index and initializing it to zero. If the original input was negative, this will cause an ArrayIndexOutOfBounds exception. This is now the trigger point.

```
for(char stonesoup_counter = 0; stonesoup_counter < stonesoup_max_char;
stonesoup_counter++) {
 output.printf("Counter value: \"%c\"\n", stonesoup_counter);
    stonesoup_array[stonesoup_counter] = 0;
}
```

If the original input was positive, or -32768, the absolute value of the input number and the char representation of it will be equivalent, the array will get initialized properly, and the snippet will execute without error.

Benign Inputs

Benign inputs are all positive inputs from 0 to 32767 inclusive, and -32768. For example:

```
1
```

```
32767
```

Since both of these inputs are within the range stated above they will not cause signed to unsigned conversion error, and are thus benign.

Exploiting Inputs

Exploiting inputs are all negative inputs from -32767 to -1 inclusive. For example:

```
1
```

```
-32767
```

Since both of these inputs are within the range stated above they will cause a signed to unsigned conversion error, and are thus exploitative.

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Restrictions and Requirements

J - CWE-196A - Unsigned to Signed Conversion Error

Summary

This snippet takes user input in the form of a number from 0 to 255 inclusive as an unsigned char, casts this number to a byte, and attempts to allocate an array using the resulting byte representation of the input. If the original input is between 128 and 255 inclusive, the conversion from an unsigned char to a signed byte will result in a negative number, leading to a NegativeArraySizeException.

This weakness differs from CWE195 because CWE195 is signed to unsigned conversion error and this weakness is unsigned to signed conversion error.

Weakness Class	Number Handling
CWE	CWE-196
Variant	A
Language	Java
Status	WEAK-133 - CWE-196-01-Java DONE
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

The snippet takes user input in the form of a number from 0 and 255 inclusive as type char (unsigned), and casts this number to a byte (signed). This is the crossover point.

```
/* STONESOUP: crossover point (user controlled value, convert from unsigned to
signed) */
int[] stonesoup_char_counts = stonesoupInitializeCounts((byte)((char)value));
```

This resulting byte is then checked to make sure it is not zero, and if not, is used to allocate an array of that size. This is the trigger point. If the original input is between 128 and 255 inclusive, the conversion from unsigned char to signed int will cause the most significant byte (one in this case) to be interpreted as a sign, and will cause the resulting number to be interpreted as negative. This will cause a

NegativeArraySizeException to be thrown.

```
if (size == 0) {
 return null;
}

Tracer.tracepointMessage("TRIGGER-POINT: BEFORE");
/* STONESOUP: trigger point (user controlled value, convert from signed to unsigned)
*/
int[] result = new int[size];
```

If the original input is between 0 and 127 inclusive, the cast will not result in a negative number, and the snippet will complete without error.

Benign Inputs

All inputs between 0 and 127 inclusive are benign. For instance:

```
2
```

```
8
```

Since both of these inputs are in the range stated above they will not cause an unsigned to signed conversion error, and are thus benign.

Exploiting Inputs

All inputs between 128 and 255 inclusive are benign. For instance:

```
-1
```

```
-10
```

Since both of these inputs are in the range stated above they will cause an unsigned to signed conversion error, and are thus exploitative.

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Restrictions and Requirements

J - CWE-197A - Numeric Truncation Error

Summary

This snippet takes a number in the range 0 to 2^{63}-1 as input, type long, and casts this number to an int, using the result as the input to SecureRandom.nextInt(). If the given input is 2,147,483,648 or larger, the cast to an int will cause a numeric truncation error resulting in a negative number which will cause the call to SecureRandom.nextInt() to throw a RuntimeException.

Weakness Class	Number Handling
CWE	CWE-197
Variant	A
Language	Java
Status	WEAK-136 - CWE-197-01-Java DONE
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

This snippet takes a number in the range 0 to 2^{63}-1 as input, type long, and casts it to an int. If the number is 2,149,483,648 or larger, the cast to an integer will result in the number being interpreted as a negative number since the most significant bit will be 1. This is the crossover point.

```
/* STONESOUP: crossover point (user controlled value, convert from long to int) */
int stonesoup_max_value = (int)((long)value);
```

This value is then used as input to the SecureRandom.nextInt() function. If the numeric truncation error has occurred, the function will throw a RuntimeException. This is the trigger point.

```
/* STONESOUP: trigger point (value is now negative, causing exception) */
output.printf("Random choice: %d\n", random.nextInt(stonesoup_max_value));
```

If the numeric truncation error did not occur, the function will execute properly and the snippet will complete without error.

Benign Inputs

All inputs between 0 and 2,149,483,647 inclusive are benign inputs. For example:

```
107
```

```
2147483647
```

Since both of these inputs are within the range stated above they will not cause a numeric truncation error and are thus benign.

Exploiting Inputs

All inputs between 2,149.483.648 and 2^{63}-1 are exploiting inputs. For example:

```
2147483648
```

```
4147483648
```

Since both of these inputs are within the range stated above they will cause a numeric truncation error and are thus exploitative.

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Restrictions and Requirements

J - CWE-369A - Divide by Zero

Summary

This snippet takes a number as input, type int from 1 to 2^{31}-1, and uses it to calculate a divisor that is later used to preform modulus division. If the input number causes the calculated divisor to equal 0, the snippet will throw a DivideByZero exception.

Weakness Class	Number Handling
CWE	CWE-369
Variant	A
Language	Java
Status	**WEAK-142** - CWE-369-01-Java **DONE**
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

This snippet takes an integer in the range 1 to 2^{31}-1 as input, checks to see if the value is equal to zero, and if not, calculates a divisor called 'random' from this input. The divisor is calculated such that, if the input is a multiple of 32768 the divisor is equal to 0. This is the crossover point.

```
if (value != 0) {
 try {
        Tracer.tracepointMessage("CROSSOVER-POINT: BEFORE");
        /* STONESOUP: crossover point (Multiple of 32768 results in zero divisor) */
        int random = (8191 * value) % (1 << 15);
```

The snippet then calculates a factor by modulo division with the divisor. If the divisor is zero, the snippet throws a DivideByZero exception. This is the trigger point.

```
/* STONESOUP: trigger point (Multiple of 32768 generates divide by zero) */
int factor = (1 << 31) % random;
```

If the divisor was not zero, the snippet will finish execution without error.

Benign Inputs

Any input between 1 and 2^{31}-1 that is not a multiple of 32768. For example:

```
245
```

```
362
```

Since neither of these inputs are a multiple of 32768, and are within the range of an int, they are benign.

Exploiting Inputs

Any multiple of 32768 that is less than 2^{31}-1. For example:

```
32768
```

```
65536
```

Since both of these inputs are a multiple of 32768, and are within the range of an int, they are exploitative.

Source Code

For more information, please refer to the weakness Source Code.

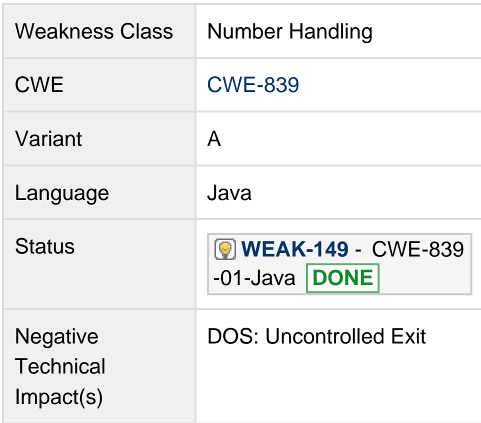
Related Information

- Execution Environment
- Restrictions and Requirements

J - CWE-839A - Numeric Range Comparison Without Minimum Check

Summary

This snippet takes a number in the range -2^{31} to 2^{31}-1, checks to make sure it is less than the size of an internal array, and if so, uses the input to access the array at that position. The snippet does not properly check the lower bound, so if the input is negative, the array access will be attempted with a negative index and the snippet will throw an ArrayIndexOutOfBounds exception.



Weakness Class	Number Handling
CWE	CWE-839
Variant	A
Language	Java
Status	**WEAK-149** - CWE-839-01-Java DONE
Negative Technical Impact(s)	DOS: Uncontrolled Exit

Implementation

The snippet takes an integer as input, declares an array of playing card face cards, and checks whether the input number is less than or equal to the size of the array. The snippet however, does not check whether or not the input number is greater than zero. This is the crossover point.

```
/* STONESOUP: crossover point (no minimum check) */
if (value >= stonesoup_face_cards.size()) {
 Tracer.tracepointMessage("CROSSOVER-POINT: AFTER");
    output.printf("Card not available for %d.\n", value);
} else {
```

If the input is less than or equal to the size of the array of face cards, the snippet uses the input value to print the corresponding face card by indexing the array with the input. However, since the input is not checked for a minimum value, the snippet will throw an ArrayOutOfBounds exception if provided with a negative input. This is the trigger point.

```
/* STONESOUP: trigger point (lookup with negative value) */
output.printf("Selected Card = %s\n", stonesoup_face_cards.get(value));
```

If the input value is non negative the snippet will execute without error.

Benign Inputs

All positive inputs within the range of int. For example:

```
2
```

```
8
```

Since both of these inputs are positive, they are benign.

Exploiting Inputs

All negative inputs within the range of int. For example:

```
-1
```

```
-10
```

Since both of these inputs are negative, they are exploitative.

Source Code

For more information, please refer to the weakness Source Code.

Related Information

- Execution Environment
- Restrictions and Requirements

Appendix A: Weakness Source Code

As a supplement to the documentation for each weakness algorithmic variant, the entire source code of each is included. Please refer to the actual source archives, as they include build scripts and the required dependencies for execution.

C Weakness Source Code

- C - CWE-120A Source Code
- C - CWE-120B Source Code
- C - CWE-120C Source Code
- C - CWE-120D Source Code
- C - CWE-124A Source Code
- C - CWE-124B Source Code
- C - CWE-124C Source Code
- C - CWE-124D Source Code
- C - CWE-126A Source Code
- C - CWE-126B Source Code
- C - CWE-126C Source Code
- C - CWE-126D Source Code
- C - CWE-127A Source Code
- C - CWE-127B Source Code
- C - CWE-127C Source Code
- C - CWE-127D Source Code
- C - CWE-129A Source Code
- C - CWE-129B Source Code
- C - CWE-134A Source Code
- C - CWE-134B Source Code
- C - CWE-170A Source Code
- C - CWE-170B Source Code
- C - CWE-190A Source Code
- C - CWE-191A Source Code
- C - CWE-191B Source Code
- C - CWE-194A Source Code
- C - CWE-195A Source Code
- C - CWE-196A Source Code
- C - CWE-197A Source Code
- C - CWE-363A Source Code
- C - CWE-367A Source Code
- C - CWE-369A Source Code
- C - CWE-412A Source Code
- C - CWE-414A Source Code
- C - CWE-415A Source Code
- C - CWE-416A Source Code
- C - CWE-476A Source Code
- C - CWE-476B Source Code
- C - CWE-476C Source Code
- C - CWE-476D Source Code
- C - CWE-476E Source Code

Java Weakness Source Code

- J - CWE-190A Source Code
- J - CWE-190B Source Code
- J - CWE-191A Source Code
- J - CWE-194A Source Code
- J - CWE-195A Source Code
- J - CWE-196A Source Code
- J - CWE-197A Source Code
- J - CWE-209A Source Code
- J - CWE-248A Source Code
- J - CWE-252A Source Code
- J - CWE-252B Source Code
- J - CWE-253A Source Code
- J - CWE-363A Source Code
- J - CWE-367A Source Code
- J - CWE-369A Source Code
- J - CWE-390A Source Code
- J - CWE-391A Source Code
- J - CWE-412A Source Code
- J - CWE-414A Source Code
- J - CWE-460A Source Code
- J - CWE-543A Source Code
- J - CWE-567A Source Code
- J - CWE-572A Source Code
- J - CWE-584A Source Code
- J - CWE-609A Source Code
- J - CWE-663A Source Code
- J - CWE-764A Source Code
- J - CWE-765A Source Code
- J - CWE-820A Source Code
- J - CWE-821A Source Code
- J - CWE-832A Source Code
- J - CWE-833A Source Code
- J - CWE-839A Source Code

Approved for public release; distribution unlimited.

- C - CWE-476F Source Code
- C - CWE-476G Source Code
- C - CWE-479A Source Code
- C - CWE-543A Source Code
- C - CWE-590A Source Code
- C - CWE-590B Source Code
- C - CWE-609A Source Code
- C - CWE-663A Source Code
- C - CWE-682A Source Code
- C - CWE-682B Source Code
- C - CWE-761A Source Code
- C - CWE-764A Source Code
- C - CWE-765A Source Code
- C - CWE-765B Source Code
- C - CWE-785A Source Code
- C - CWE-785B Source Code
- C - CWE-785C Source Code
- C - CWE-785D Source Code
- C - CWE-805A Source Code
- C - CWE-805B Source Code
- C - CWE-805C Source Code
- C - CWE-805D Source Code
- C - CWE-806A Source Code
- C - CWE-806B Source Code
- C - CWE-806C Source Code
- C - CWE-806D Source Code
- C - CWE-820A Source Code
- C - CWE-821A Source Code
- C - CWE-822A Source Code
- C - CWE-824A Source Code
- C - CWE-824B Source Code
- C - CWE-828A Source Code
- C - CWE-831A Source Code
- C - CWE-833A Source Code
- C - CWE-839A Source Code
- C - CWE-843A Source Code

C Weakness Source Code

A collection of the source code for each C weakness algorithmic variant.

Concurrency Handling Source

- C - CWE-363A Source Code
- C - CWE-367A Source Code
- C - CWE-412A Source Code
- C - CWE-414A Source Code

Injection Source

Content by label

There is no content with the specified labels

Memory Corruption Source

- C - CWE-120A Source Code
- C - CWE-120B Source Code
- C - CWE-120C Source Code
- C - CWE-120D Source Code

- C - CWE-479A Source Code
- C - CWE-543A Source Code
- C - CWE-609A Source Code
- C - CWE-663A Source Code
- C - CWE-764A Source Code
- C - CWE-765A Source Code
- C - CWE-765B Source Code
- C - CWE-820A Source Code
- C - CWE-821A Source Code
- C - CWE-828A Source Code
- C - CWE-831A Source Code
- C - CWE-833A Source Code

- C - CWE-124A Source Code
- C - CWE-124B Source Code
- C - CWE-124C Source Code
- C - CWE-124D Source Code
- C - CWE-126A Source Code
- C - CWE-126B Source Code
- C - CWE-126C Source Code
- C - CWE-126D Source Code
- C - CWE-127A Source Code
- C - CWE-127B Source Code
- C - CWE-127C Source Code
- C - CWE-127D Source Code
- C - CWE-129A Source Code
- C - CWE-129B Source Code
- C - CWE-134A Source Code
- C - CWE-134B Source Code
- C - CWE-170A Source Code
- C - CWE-170B Source Code
- C - CWE-415A Source Code
- C - CWE-416A Source Code
- C - CWE-590A Source Code
- C - CWE-590B Source Code
- C - CWE-761A Source Code
- C - CWE-785A Source Code
- C - CWE-785B Source Code
- C - CWE-785C Source Code

43 related results

Null Pointer Source

- C - CWE-476A Source Code
- C - CWE-476B Source Code
- C - CWE-476C Source Code
- C - CWE-476D Source Code
- C - CWE-476E Source Code
- C - CWE-476F Source Code
- C - CWE-476G Source Code

Number Handling Source

- C - CWE-190A Source Code
- C - CWE-191A Source Code
- C - CWE-191B Source Code
- C - CWE-194A Source Code
- C - CWE-195A Source Code
- C - CWE-196A Source Code
- C - CWE-197A Source Code
- C - CWE-369A Source Code
- C - CWE-682A Source Code
- C - CWE-682B Source Code
- C - CWE-839A Source Code

Resource Drains Source

Content by label

There is no content with the specified labels

C - CWE-120A Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
```

```
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1
#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1
#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1
#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1
#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1
#endif /* NOTHING DEFINED */

#if STONESOUP_SNIPPET_INCLUDES
/*******************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 *******************************************************************************/
#include <stdio.h>  /* printf */
#include <stdlib.h> /* malloc */
#include <string.h> /* memset */
/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_INCLUDES */
#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 *******************************************************************************/
#include "output-handlers/handler.h"
#endif /* STONESOUP_SNIPPET_DEBUG */
int stonesoup_toupper(int c) {
    if (c >= 97 && c <= 122) {
        return c - 32;
    }
    return c;
}
#endif /* STONESOUP_SNIPPET_SUPPORT */

#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
```

```
/*****************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 *****************************************************************************/
void weakness(char * stonesoup_tainted_buff) {
#define STONESOUP_TAINT_SOURCE stonesoup_tainted_buff
#endif /* STONESOUP_SNIPPET_DEBUG */
#if STONESOUP_SNIPPET_BODY_DECLARATIONS
    int stonesoup_oc_i = 0;
 char stonesoup_stack_buffer_64[64];
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_start, "CWE120", "A", "Buffer Copy without
Checking Size of Input");
#endif /* STONESOUP_TRACE */
 memset(stonesoup_stack_buffer_64,0,64);
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, variable_signed_integral, "stonesoup_oc_i",
stonesoup_oc_i, &stonesoup_oc_i, "INITIAL-STATE");
    tracepoint(stonesoup_trace, variable_buffer, "stonesoup_stack_buffer_64",
stonesoup_stack_buffer_64, "INITIAL-STATE");
    tracepoint(stonesoup_trace, variable_address, "__builtin_return_address(0)",
__builtin_return_address(0), "INITIAL-STATE");
    tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */

 /* STONESOUP: CROSSOVER-POINT (Unchecked buffer copy) */
 strcpy(stonesoup_stack_buffer_64,STONESOUP_TAINT_SOURCE);
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, variable_buffer, "stonesoup_stack_buffer_64",
stonesoup_stack_buffer_64, "CROSSOVER-STATE");
    tracepoint(stonesoup_trace, variable_address, "__builtin_return_address(0)",
__builtin_return_address(0), "CROSSOVER-STATE");
    tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
#endif /* STONESOUP_TRACE */

    for (; stonesoup_oc_i < 64; ++stonesoup_oc_i) {
        stonesoup_stack_buffer_64[stonesoup_oc_i] =
stonesoup_toupper(stonesoup_stack_buffer_64[stonesoup_oc_i]);
    }

#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, variable_signed_integral, "stonesoup_oc_i",
stonesoup_oc_i, &stonesoup_oc_i, "FINAL-STATE");
    tracepoint(stonesoup_trace, variable_buffer, "stonesoup_stack_buffer_64",
stonesoup_stack_buffer_64, "FINAL-STATE");
#endif /* STONESOUP_TRACE */

 stonesoup_printf("%s\n",stonesoup_stack_buffer_64);
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
    tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
```

```
    /* STONESOUP: TRIGGER-POINT (Buffer Overflow: Unchecked stack buffer copy) */
    /* Trigger point occurs on function return. */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 ******************************************************************************/
int main(int argc,char *argv[])
{
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    char* tainted_buff = argv[1];

    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
    return 0;
}
```

```
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-120B Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1
#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1
#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1
#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1
#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1
#endif /* NOTHING DEFINED */

#if STONESOUP_SNIPPET_INCLUDES
/*****************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 ****************************************************************************/
#include <stdlib.h> /* malloc */
#include <stdio.h>  /* fprintf */
#include <string.h> /* memset */
#include <errno.h>  /* errno */
/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_INCLUDES */
#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 ****************************************************************************/
#include "output-handlers/handler.h"
```

```
#endif /* STONESOUP_SNIPPET_DEBUG */
int stonesoup_toupper(int c)
{
  if (c >= 97 && c <= 122) {
    return c - 32;
  }
  return c;
}
#endif /* STONESOUP_SNIPPET_SUPPORT */

#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 *****************************************************************************/
void weakness(char * stonesoup_tainted_buff) {
#define STONESOUP_TAINT_SOURCE stonesoup_tainted_buff
#endif /* STONESOUP_SNIPPET_DEBUG */
#if STONESOUP_SNIPPET_BODY_DECLARATIONS
    int stonesoup_i = 0;
    int stonesoup_opt_var;
    char *stonesoup_heap_buffer_64 = 0;
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_start, "CWE120", "B", "Buffer Copy without
Checking Size of Input");
#endif /* STONESOUP_TRACE */
    stonesoup_heap_buffer_64 = (char*) malloc(64 * sizeof(char));
    if (stonesoup_heap_buffer_64 != NULL) {
        memset(stonesoup_heap_buffer_64,0,64);

#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, variable_signed_integral, "stonesoup_i",
stonesoup_i, &stonesoup_i, "INITIAL-STATE");
        tracepoint(stonesoup_trace, variable_buffer, "stonesoup_heap_buffer_64",
stonesoup_heap_buffer_64, "INITIAL-STATE");
        tracepoint(stonesoup_trace, variable_address, "__builtin_return_address(0)",
__builtin_return_address(0), "INITIAL-STATE");
        tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
    /* STONESOUP: CROSSOVER-POINT (Unchecked buffer copy) */
    /* STONESOUP: TRIGGER-POINT (Buffer Overflow: Unchecked heap buffer copy) */
        strcpy(stonesoup_heap_buffer_64, STONESOUP_TAINT_SOURCE);

#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, variable_buffer, "stonesoup_heap_buffer_64",
stonesoup_heap_buffer_64, "CROSSOVER-STATE");
        tracepoint(stonesoup_trace, variable_address, "__builtin_return_address(0)",
__builtin_return_address(0), "CROSSOVER-STATE");
        tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
        stonesoup_opt_var = strlen( stonesoup_heap_buffer_64);
```

```
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, variable_signed_integral, "stonesoup_opt_var",
stonesoup_opt_var, &stonesoup_opt_var, "CROSSOVER-STATE");
#endif /* STONESOUP_TRACE */

        for (; stonesoup_i < stonesoup_opt_var; ++stonesoup_i) {
            stonesoup_heap_buffer_64[stonesoup_i] =
stonesoup_toupper(stonesoup_heap_buffer_64[stonesoup_i]);
        }
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, variable_signed_integral, "stonesoup_i",
stonesoup_i, &stonesoup_i, "FINAL-STATE");
        tracepoint(stonesoup_trace, variable_signed_integral, "stonesoup_opt_var",
stonesoup_opt_var, &stonesoup_opt_var, "FINAL-STATE");
        tracepoint(stonesoup_trace, variable_buffer, "stonesoup_heap_buffer_64",
stonesoup_heap_buffer_64, "BEFORE-FREE");
#endif /* STONESOUP_TRACE */
        stonesoup_printf("%s\n",stonesoup_heap_buffer_64);

        free(stonesoup_heap_buffer_64);
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, variable_buffer, "stonesoup_heap_buffer_64",
stonesoup_heap_buffer_64, "FINAL-STATE");
        tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
        tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
    }
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_DEBUG
/******************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 ******************************************************************************/
int main(int argc,char *argv[])
{
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    char* tainted_buff = argv[1];

    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }

    return 0;
}
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-120C Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1
#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1
#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1
#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1
#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1
#endif /* NOTHING DEFINED */

#if STONESOUP_SNIPPET_INCLUDES
/*****************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 *****************************************************************************/
#include <stdlib.h> /* malloc */
#include <stdio.h>  /* fprintf */
#include <string.h> /* memset */
#include <errno.h>  /* errno */
/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_INCLUDES */
#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 *****************************************************************************/
#include "output-handlers/handler.h"
```

```
#endif /* STONESOUP_SNIPPET_DEBUG */
int stonesoup_toupper(int c)
{
  if (c >= 97 && c <= 122) {
    return c - 32;
  }
  return c;
}
struct stonesoup_struct {
    int (* before)(int);
    char buffer[64];
    int (* after)(int);
};

#endif /* STONESOUP_SNIPPET_SUPPORT */

#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 ******************************************************************************/
void weakness(char * stonesoup_tainted_buff) {
#define STONESOUP_TAINT_SOURCE stonesoup_tainted_buff
#endif /* STONESOUP_SNIPPET_DEBUG */
#if STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_start, "CWE120", "C", "Buffer Copy without 
Checking Size of Input");
#endif /* STONESOUP_TRACE */
    int stonesoup_i = 0;
    int stonesoup_opt_var;
    struct stonesoup_struct* stonesoup_data = NULL;
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, variable_signed_integral, "stonesoup_i", stonesoup_i, 
&stonesoup_i, "INITIAL-STATE");
#endif /* STONESOUP_TRACE */
    stonesoup_data = (struct stonesoup_struct*) malloc(sizeof(struct 
stonesoup_struct));
    if (stonesoup_data != NULL) {
        stonesoup_data->before = stonesoup_toupper;
        memset(stonesoup_data->buffer,0,64);
        stonesoup_data->after = stonesoup_toupper;
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, variable_signed_integral, 
"stonesoup_data->before", stonesoup_data->before, &stonesoup_data->before, 
"INITIAL-STATE");
        tracepoint(stonesoup_trace, variable_buffer, "stonesoup_data->buffer", 
stonesoup_data->buffer, "INITIAL-STATE");
        tracepoint(stonesoup_trace, variable_signed_integral, "stonesoup_data->after", 
stonesoup_data->after, &stonesoup_data->after, "INITIAL-STATE");
        tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
```

```
        tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */

    /* STONESOUP: CROSSOVER-POINT (Unchecked buffer copy) */
    /* STONESOUP: TRIGGER-POINT (Buffer Overflow: Unchecked heap buffer copy) */
        strcpy(stonesoup_data->buffer, STONESOUP_TAINT_SOURCE);
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, variable_buffer, "stonesoup_data->buffer",
stonesoup_data->buffer, "CROSSOVER-STATE");
        tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
        stonesoup_opt_var = strlen( stonesoup_data->buffer);
        for (; stonesoup_i < stonesoup_opt_var; ++stonesoup_i) {
            stonesoup_data->buffer[stonesoup_i] =
stonesoup_toupper(stonesoup_data->buffer[stonesoup_i]);

stonesoup_printf("%c",stonesoup_data->after(stonesoup_data->buffer[stonesoup_i]));
        }
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, variable_signed_integral, "stonesoup_i",
stonesoup_i, &stonesoup_i, "FINAL-STATE");
        tracepoint(stonesoup_trace, variable_buffer, "stonesoup_data->buffer",
stonesoup_data->buffer, "FINAL-STATE");
#endif /* STONESOUP_TRACE */

        stonesoup_printf("\n");

        free(stonesoup_data);

#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
    }
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 *****************************************************************************/
int main(int argc,char *argv[])
{
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    char* tainted_buff = argv[1];

    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
```

```
    return 0;
}
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-120D Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1
#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1
#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1
#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1
#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1
#endif /* NOTHING DEFINED */

#if STONESOUP_SNIPPET_INCLUDES
/*****************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 *****************************************************************************/
#include <stdlib.h> /* malloc */
#include <stdio.h>  /* fprintf */
#include <string.h> /* memset */
#include <errno.h>  /* errno */
/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_INCLUDES */
#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 *****************************************************************************/
```

```
#include "output-handlers/handler.h"
#endif /* STONESOUP_SNIPPET_DEBUG */
int stonesoup_toupper(int c)
{
  if (c >= 97 && c <= 122) {
    return c - 32;
  }
  return c;
}
struct stonesoup_struct {
    int (* before)(int);
    char buffer[64];
    int (* after)(int);
};

#endif /* STONESOUP_SNIPPET_SUPPORT */

#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 *******************************************************************************/
void weakness(char * stonesoup_tainted_buff) {
#define STONESOUP_TAINT_SOURCE stonesoup_tainted_buff
#endif /* STONESOUP_SNIPPET_DEBUG */
#if STONESOUP_SNIPPET_BODY_DECLARATIONS
    int stonesoup_i = 0;
    int stonesoup_opt_var;
    struct stonesoup_struct stonesoup_data;
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_start, "CWE120", "D", "Buffer Copy without
Checking Size of Input");
#endif /* STONESOUP_TRACE */
    stonesoup_data.before = stonesoup_toupper;
    for (stonesoup_i = 0; stonesoup_i < 64; stonesoup_i++) {
        stonesoup_data.buffer[stonesoup_i] = 0;
    }
    stonesoup_data.after = stonesoup_toupper;
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, variable_signed_integral, "stonesoup_i", stonesoup_i,
&stonesoup_i, "INITIAL-STATE");
    tracepoint(stonesoup_trace, variable_signed_integral, "stonesoup_data.before",
stonesoup_data.before, &stonesoup_data.before, "INITIAL-STATE");
    tracepoint(stonesoup_trace, variable_buffer, "stonesoup_data.buffer",
stonesoup_data.buffer, "INITIAL-STATE");
    tracepoint(stonesoup_trace, variable_signed_integral, "stonesoup_data.after",
stonesoup_data.after, &stonesoup_data.after, "INITIAL-STATE");
    tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
    tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
/* STONESOUP: CROSSOVER-POINT (Unchecked buffer copy) */
```

```
    strcpy(stonesoup_data.buffer, STONESOUP_TAINT_SOURCE);
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, variable_buffer, "stonesoup_data.buffer",
stonesoup_data.buffer, "CROSSOVER-STATE");
    tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
/* STONESOUP: TRIGGER-POINT (Buffer Overflow: Unchecked heap buffer copy) */
    stonesoup_opt_var = strlen( stonesoup_data.buffer);
    for (stonesoup_i = 0; stonesoup_i < stonesoup_opt_var; ++stonesoup_i) {
        stonesoup_data.buffer[stonesoup_i] =
stonesoup_toupper(stonesoup_data.buffer[stonesoup_i]);

stonesoup_printf("%c",stonesoup_data.after(stonesoup_data.buffer[stonesoup_i]));
    }
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, variable_signed_integral, "stonesoup_i", stonesoup_i,
&stonesoup_i, "FINAL-STATE");
    tracepoint(stonesoup_trace, variable_buffer, "stonesoup_data.buffer",
stonesoup_data.buffer, "FINAL-STATE");
#endif /* STONESOUP_TRACE */
    stonesoup_printf("\n");

#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_DEBUG
/******************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 ******************************************************************************/
int main(int argc,char *argv[])
{
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    char* tainted_buff = argv[1];

    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }

    return 0;
}
```

```
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-124A Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1
#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1
#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1
#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1
#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1
#endif /* NOTHING DEFINED */

#if STONESOUP_SNIPPET_INCLUDES
/******************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 ******************************************************************************/
#include <stdlib.h> /* malloc */
#include <stdio.h>  /* fprintf */
#include <string.h> /* memset */
#include <errno.h>  /* errno */
/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_INCLUDES */
#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
/******************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 ******************************************************************************/
#include "output-handlers/handler.h"
```

```
#endif /* STONESOUP_SNIPPET_DEBUG */
int stonesoup_toupper(int c)
{
  if (c >= 97 && c <= 122) {
    return c - 32;
  }
  return c;
}
#endif /* STONESOUP_SNIPPET_SUPPORT */

#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 *****************************************************************************/
void weakness(char * stonesoup_tainted_buff) {
#define STONESOUP_TAINT_SOURCE stonesoup_tainted_buff
#endif /* STONESOUP_SNIPPET_DEBUG */
#if STONESOUP_SNIPPET_BODY_DECLARATIONS
 int stonesoup_oc_i = 0;
 char stonesoup_stack_buff_64[64];
 char *stonesoup_other_buff[8];
 int stonesoup_my_buff_size = 63;
 int stonesoup_buff_size;
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_start, "CWE124", "A", "Buffer Underwrite");
    tracepoint(stonesoup_trace, variable_signed_integral, "stonesoup_oc_i",
stonesoup_oc_i, &stonesoup_oc_i, "INITIAL-STATE");
    tracepoint(stonesoup_trace, variable_signed_integral, "stonesoup_my_buff_size",
stonesoup_my_buff_size, &stonesoup_my_buff_size, "INITIAL-STATE");
#endif /* STONESOUP_TRACE */
 stonesoup_other_buff[7] = STONESOUP_TAINT_SOURCE;
    memset(stonesoup_stack_buff_64,65,64);
    stonesoup_stack_buff_64[64 - 1] = '\0';
    stonesoup_buff_size = ((int )(strlen(STONESOUP_TAINT_SOURCE)));
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, variable_buffer, "stonesoup_stack_buff_64",
stonesoup_stack_buff_64, "INITIAL-STATE");
    tracepoint(stonesoup_trace, variable_buffer, "stonesoup_other_buff",
stonesoup_other_buff, "INITIAL-STATE");
    tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
    tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
    for (; stonesoup_buff_size >= 0; (--stonesoup_my_buff_size ,
--stonesoup_buff_size)) {
  /* STONESOUP: CROSSOVER-POINT (Buffer Underwrite) */
  /* STONESOUP: TRIGGER-POINT (Buffer Underwrite: Stack Underflow) */
        stonesoup_stack_buff_64[stonesoup_my_buff_size] =
STONESOUP_TAINT_SOURCE[stonesoup_buff_size];
    }
#if STONESOUP_TRACE
```

```
    tracepoint(stonesoup_trace, variable_buffer, "stonesoup_stack_buff_64",
stonesoup_stack_buff_64, "CROSSOVER-STATE");
#endif /* STONESOUP_TRACE */
    for (; stonesoup_oc_i < 64; ++stonesoup_oc_i) {
        stonesoup_stack_buff_64[stonesoup_oc_i] =
stonesoup_toupper(stonesoup_stack_buff_64[stonesoup_oc_i]);
    }
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: AFTER");
    tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
    stonesoup_printf("%s\n",stonesoup_stack_buff_64);
    stonesoup_printf("strlen size = %d\n",strlen(STONESOUP_TAINT_SOURCE));
    stonesoup_printf("strlen size = %d\n",strlen(stonesoup_other_buff[7]));
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, variable_signed_integral, "stonesoup_oc_i",
stonesoup_oc_i, &stonesoup_oc_i, "FINAL-STATE");
    tracepoint(stonesoup_trace, variable_buffer, "stonesoup_stack_buff_64",
stonesoup_stack_buff_64, "FINAL-STATE");
    tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_DEBUG
/******************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 *****************************************************************************/
int main(int argc,char *argv[])
{
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    char* tainted_buff = argv[1];
    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
    return 0;
}
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-124B Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1
#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1
#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1
#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1
#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1
#endif /* NOTHING DEFINED */

#if STONESOUP_SNIPPET_INCLUDES
/*****************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 *****************************************************************************/
#include <stdlib.h> /* malloc */
#include <stdio.h>  /* fprintf */
#include <string.h> /* memset */
#include <errno.h>  /* errno */
/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_INCLUDES */
#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 *****************************************************************************/
#include "output-handlers/handler.h"
```

```
#endif /* STONESOUP_SNIPPET_DEBUG */
int stonesoup_toupper(int c)
{
  if (c >= 97 && c <= 122) {
    return c - 32;
  }
  return c;
}
#endif /* STONESOUP_SNIPPET_SUPPORT */

#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/******************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 ******************************************************************************/
void weakness(char * stonesoup_tainted_buff) {
#define STONESOUP_TAINT_SOURCE stonesoup_tainted_buff
#endif /* STONESOUP_SNIPPET_DEBUG */
#if STONESOUP_SNIPPET_BODY_DECLARATIONS
    int stonesoup_i = 0;
    int stonesoup_buff_size = 63;
    int stonesoup_taint_len;
    char *stonesoup_heap_buff_64 = 0;
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_start, "CWE124", "B", "Buffer Underwrite");
    tracepoint(stonesoup_trace, variable_signed_integral, "stonesoup_i", stonesoup_i,
&stonesoup_i, "INITIAL-STATE");
    tracepoint(stonesoup_trace, variable_signed_integral, "stonesoup_buff_size",
stonesoup_buff_size, &stonesoup_buff_size, "INITIAL-STATE");
#endif /* STONESOUP_TRACE */
    stonesoup_heap_buff_64 = (char*) malloc(64 * sizeof(char));
    if (stonesoup_heap_buff_64 != NULL) {
        memset(stonesoup_heap_buff_64, 'A' ,64);
        stonesoup_heap_buff_64[64 - 1] = '\0';
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, variable_buffer, "stonesoup_heap_buff_64",
stonesoup_heap_buff_64, "INITIAL-STATE");
        tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
        tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
        stonesoup_taint_len = ((int )(strlen(STONESOUP_TAINT_SOURCE)));
        for (; stonesoup_taint_len >= 0; (--stonesoup_buff_size ,
--stonesoup_taint_len)) {
    /* STONESOUP: CROSSOVER-POINT (Buffer Underwrite) */
    /* STONESOUP: TRIGGER-POINT (Buffer Underwrite: Heap Underflow) */
            stonesoup_heap_buff_64[stonesoup_buff_size] =
STONESOUP_TAINT_SOURCE[stonesoup_taint_len];
        }
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, variable_buffer, "stonesoup_heap_buff_64",
stonesoup_heap_buff_64, "CROSSOVER-STATE");
```

Approved for public release; distribution unlimited.

```
#endif /* STONESOUP_TRACE */
        for (; stonesoup_i < 64; ++stonesoup_i) {
            stonesoup_heap_buff_64[stonesoup_i] =
stonesoup_toupper(stonesoup_heap_buff_64[stonesoup_i]);
        }
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: AFTER");
        tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
        stonesoup_printf("%s\n",&(stonesoup_heap_buff_64[stonesoup_buff_size+1]));
        free(stonesoup_heap_buff_64);
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, variable_signed_integral, "stonesoup_i",
stonesoup_i, &stonesoup_i, "FINAL-STATE");
        tracepoint(stonesoup_trace, variable_buffer, "stonesoup_heap_buff_64",
stonesoup_heap_buff_64, "FINAL-STATE");
        tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
    }
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 *******************************************************************************/
int main(int argc,char *argv[])
{
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    char* tainted_buff = argv[1];
    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
    return 0;
}
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-124C Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1
#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1
#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1
#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1
#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1
#endif /* NOTHING DEFINED */

#if STONESOUP_SNIPPET_INCLUDES
/*****************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 *****************************************************************************/
#include <stdlib.h> /* malloc */
#include <stdio.h>  /* fprintf */
#include <string.h> /* memset */
#include <errno.h>  /* errno */
/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_INCLUDES */
#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 *****************************************************************************/
#include "output-handlers/handler.h"
```

Approved for public release; distribution unlimited.

```
#endif /* STONESOUP_SNIPPET_DEBUG */
int stonesoup_toupper(int c)
{
  if (c >= 97 && c <= 122) {
    return c - 32;
  }
  return c;
}
struct stonesoup_struct {
    int (* before)(int);
    char buffer[64];
    int (* after)(int);
};
#endif /* STONESOUP_SNIPPET_SUPPORT */

#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 ******************************************************************************/
void weakness(char * stonesoup_tainted_buff) {
#define STONESOUP_TAINT_SOURCE stonesoup_tainted_buff
#endif /* STONESOUP_SNIPPET_DEBUG */
#if STONESOUP_SNIPPET_BODY_DECLARATIONS
    int stonesoup_i = 0;
    struct stonesoup_struct* stonesoup_data = NULL;
    int stonesoup_buff_size = 63;
    int stonesoup_taint_len;
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_start, "CWE124", "C", "Buffer Underwrite");
    tracepoint(stonesoup_trace, variable_signed_integral, "stonesoup_i", stonesoup_i,
&stonesoup_i, "INITIAL-STATE");
    tracepoint(stonesoup_trace, variable_signed_integral, "stonesoup_buff_size",
stonesoup_buff_size, &stonesoup_buff_size, "INITIAL-STATE");
#endif /* STONESOUP_TRACE */
    stonesoup_data = (struct stonesoup_struct*) malloc(sizeof(struct
stonesoup_struct));
    if (stonesoup_data != NULL) {
        stonesoup_data->before = stonesoup_toupper;
        memset(stonesoup_data->buffer, 'A', 64);
        stonesoup_data->buffer[63] = '\0';
        stonesoup_data->after = stonesoup_toupper;
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, variable_signed_integral,
"stonesoup_data->before", stonesoup_data->before, &stonesoup_data->before,
"INITIAL-STATE");
        tracepoint(stonesoup_trace, variable_buffer, "stonesoup_data->buffer",
stonesoup_data->buffer, "INITIAL-STATE");
        tracepoint(stonesoup_trace, variable_signed_integral, "stonesoup_data->after",
stonesoup_data->after, &stonesoup_data->after, "INITIAL-STATE");
        tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
```

```
        tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
        stonesoup_taint_len = ((int )(strlen(STONESOUP_TAINT_SOURCE)));
    /* STONESOUP: CROSSOVER-POINT (Buffer Underwrite) */
    /* STONESOUP: TRIGGER-POINT (Buffer Underwrite: Heap Underflow) */
        for (; stonesoup_taint_len >= 0; (--stonesoup_buff_size ,
--stonesoup_taint_len)) {
            stonesoup_data->buffer[stonesoup_buff_size] =
STONESOUP_TAINT_SOURCE[stonesoup_taint_len];
        }
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, variable_buffer, "stonesoup_data->buffer",
stonesoup_data->buffer, "CROSSOVER-STATE");
#endif /* STONESOUP_TRACE */
        for (stonesoup_i = 0; stonesoup_i < 64; ++stonesoup_i) {
            stonesoup_data->buffer[stonesoup_i] =
stonesoup_toupper(stonesoup_data->buffer[stonesoup_i]);

stonesoup_printf("%c",stonesoup_data->before(stonesoup_data->buffer[stonesoup_i]));
        }
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: AFTER");
        tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
        stonesoup_printf("\n");
        free(stonesoup_data);
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, variable_signed_integral, "stonesoup_i",
stonesoup_i, &stonesoup_i, "FINAL-STATE");
        tracepoint(stonesoup_trace, variable_buffer, "stonesoup_data->buffer",
stonesoup_data->buffer, "FINAL-STATE");
        tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
    }
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 ******************************************************************************/
int main(int argc,char *argv[])
{
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    char* tainted_buff = argv[1];
    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
    return 0;
}
```

```
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-124D Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1
#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1
#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1
#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1
#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1
#endif /* NOTHING DEFINED */

#if STONESOUP_SNIPPET_INCLUDES
/*****************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 *****************************************************************************/
#include <stdlib.h> /* malloc */
#include <stdio.h>  /* fprintf */
#include <string.h> /* memset */
#include <errno.h>  /* errno */
/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_INCLUDES */
#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 *****************************************************************************/
#include "output-handlers/handler.h"
```

```
#endif /* STONESOUP_SNIPPET_DEBUG */
int stonesoup_toupper(int c)
{
  if (c >= 97 && c <= 122) {
    return c - 32;
  }
  return c;
}
struct stonesoup_struct {
    int (* before)(int);
    char buffer[64];
    int (* after)(int);
};
#endif /* STONESOUP_SNIPPET_SUPPORT */

#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 *****************************************************************************/
void weakness(char * stonesoup_tainted_buff) {
#define STONESOUP_TAINT_SOURCE stonesoup_tainted_buff
#endif /* STONESOUP_SNIPPET_DEBUG */
#if STONESOUP_SNIPPET_BODY_DECLARATIONS
    int stonesoup_i = 0;
    struct stonesoup_struct stonesoup_data;
    int stonesoup_buff_size = 63;
    int stonesoup_taint_len;
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_start, "CWE124", "D", "Buffer Underwrite");
    tracepoint(stonesoup_trace, variable_signed_integral, "stonesoup_i", stonesoup_i,
&stonesoup_i, "INITIAL-STATE");
    tracepoint(stonesoup_trace, variable_signed_integral, "stonesoup_buff_size",
stonesoup_buff_size, &stonesoup_buff_size, "INITIAL-STATE");
#endif /* STONESOUP_TRACE */
    stonesoup_data.before = stonesoup_toupper;
    for (stonesoup_i = 0; stonesoup_i < 64; stonesoup_i++) {
        stonesoup_data.buffer[stonesoup_i] = 'A';
    }
    stonesoup_data.buffer[63] = '\0';
    stonesoup_data.after = stonesoup_toupper;
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, variable_signed_integral, "stonesoup_data.before",
stonesoup_data.before, &stonesoup_data.before, "INITIAL-STATE");
    tracepoint(stonesoup_trace, variable_buffer, "stonesoup_data.buffer",
stonesoup_data.buffer, "INITIAL-STATE");
    tracepoint(stonesoup_trace, variable_signed_integral, "stonesoup_data.after",
stonesoup_data.after, &stonesoup_data.after, "INITIAL-STATE");
    tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
    tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
```

```
    stonesoup_taint_len = ((int )(strlen(STONESOUP_TAINT_SOURCE)));
/* STONESOUP: CROSSOVER-POINT (Buffer Underwrite) */
/* STONESOUP: TRIGGER-POINT (Buffer Underwrite: Heap Underflow) */
    for (; stonesoup_taint_len >= 0; (--stonesoup_buff_size , --stonesoup_taint_len))
{
        stonesoup_data.buffer[stonesoup_buff_size] =
STONESOUP_TAINT_SOURCE[stonesoup_taint_len];
    }
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, variable_buffer, "stonesoup_data.buffer",
stonesoup_data.buffer, "CROSSOVER-STATE");
#endif /* STONESOUP_TRACE */
    for (stonesoup_i = 0; stonesoup_i < 64; ++stonesoup_i) {
        stonesoup_data.buffer[stonesoup_i] =
stonesoup_toupper(stonesoup_data.buffer[stonesoup_i]);

stonesoup_printf("%c",stonesoup_data.before(stonesoup_data.buffer[stonesoup_i]));
    }
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: AFTER");
    tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
    stonesoup_printf("\n");
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, variable_signed_integral, "stonesoup_i", stonesoup_i,
&stonesoup_i, "FINAL-STATE");
    tracepoint(stonesoup_trace, variable_buffer, "stonesoup_data.buffer",
stonesoup_data.buffer, "FINAL-STATE");
    tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 *******************************************************************************/
int main(int argc,char *argv[])
{
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    char* tainted_buff = argv[1];
    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
    return 0;
}
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-126A Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1
#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1
#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1
#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1
#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1
#endif /* NOTHING DEFINED */

#if STONESOUP_SNIPPET_INCLUDES
/*****************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 *****************************************************************************/
#include <stdlib.h> /* malloc */
#include <stdio.h> /* fprintf */
#include <string.h> /* memset */
#include <errno.h> /* errno */
/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_INCLUDES */
#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 *****************************************************************************/
#include "output-handlers/handler.h"
```

```
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_SUPPORT */

#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 *****************************************************************************/
void weakness(char * stonesoup_tainted_buff) {
#define STONESOUP_TAINT_SOURCE stonesoup_tainted_buff
#endif /* STONESOUP_SNIPPET_DEBUG */
#if STONESOUP_SNIPPET_BODY_DECLARATIONS
 int stonesoup_ss_i = 0;
 char stonesoup_stack_buff_64[64];
 int stonesoup_my_buff_size;
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_start, "CWE126", "A", "Buffer Over-read");
    tracepoint(stonesoup_trace, variable_signed_integral, "stonesoup_ss_i",
stonesoup_ss_i, &stonesoup_ss_i, "INITIAL-STATE");
#endif /* STONESOUP_TRACE */
 memset(stonesoup_stack_buff_64,'A',63);
 stonesoup_stack_buff_64[63] = '\0';

#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, variable_buffer, "stonesoup_stack_buff_64",
stonesoup_stack_buff_64, "INITIAL-STATE");
    tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
    tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */

    stonesoup_my_buff_size = ((int )(strlen(STONESOUP_TAINT_SOURCE)));
 for (; stonesoup_ss_i < stonesoup_my_buff_size; ++stonesoup_ss_i){
  /* STONESOUP: CROSSOVER-POINT (Buffer Overread) */
  /* STONESOUP: TRIGGER-POINT (Buffer Overread) */
  stonesoup_printf("%c",stonesoup_stack_buff_64[stonesoup_ss_i]);
 }

#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, variable_buffer, "stonesoup_stack_buff_64",
stonesoup_stack_buff_64, "CROSSOVER-STATE");
    tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
    tracepoint(stonesoup_trace, variable_signed_integral, "stonesoup_ss_i",
stonesoup_ss_i, &stonesoup_ss_i, "FINAL-STATE");
    tracepoint(stonesoup_trace, variable_buffer, "stonesoup_stack_buff_64",
stonesoup_stack_buff_64, "FINAL-STATE");
#endif /* STONESOUP_TRACE */
 stonesoup_printf("\n");
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
```

```
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 *****************************************************************************/
int main(int argc,char *argv[])
{
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    char* tainted_buff = argv[1];

    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
    return 0;
}
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-126B Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1
#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1
#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1
#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1
#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1
#endif /* NOTHING DEFINED */

#if STONESOUP_SNIPPET_INCLUDES
/*******************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 *******************************************************************************/
#include <stdlib.h> /* malloc */
#include <stdio.h> /* fprintf */
#include <string.h> /* memset */
#include <errno.h> /* errno */
/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_INCLUDES */
#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 *******************************************************************************/
#include "output-handlers/handler.h"
```

```
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_SUPPORT */

#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 ******************************************************************************/
void weakness(char * stonesoup_tainted_buff) {
#define STONESOUP_TAINT_SOURCE stonesoup_tainted_buff
#endif /* STONESOUP_SNIPPET_DEBUG */
#if STONESOUP_SNIPPET_BODY_DECLARATIONS
    int stonesoup_ss_i = 0;
    char* stonesoup_heap_buff_64;
    int stonesoup_buff_size;
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_start, "CWE126", "B", "Buffer Over-read");
#endif /* STONESOUP_TRACE */
    stonesoup_heap_buff_64 = (char*) malloc(64 * sizeof(char));
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, variable_signed_integral, "stonesoup_ss_i",
stonesoup_ss_i, &stonesoup_ss_i, "INITIAL-STATE");
#endif /* STONESOUP_TRACE */
    if (stonesoup_heap_buff_64 != NULL) {
        memset(stonesoup_heap_buff_64,'A',63);
        stonesoup_heap_buff_64[63] = '\0';
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, variable_buffer, "stonesoup_heap_buff_64",
stonesoup_heap_buff_64, "INITIAL-STATE");
#endif /* STONESOUP_TRACE */
        stonesoup_buff_size = ((int )(strlen(STONESOUP_TAINT_SOURCE)));
        strncpy(stonesoup_heap_buff_64, STONESOUP_TAINT_SOURCE, 64);
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, variable_buffer, "stonesoup_heap_buff_64",
stonesoup_heap_buff_64, "TAINTED");
        tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
        tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
        for (; stonesoup_ss_i < stonesoup_buff_size; ++stonesoup_ss_i){
            /* STONESOUP: CROSSOVER-POINT (Buffer Overread) */
            /* STONESOUP: TRIGGER-POINT (Buffer Overread) */
            stonesoup_printf("%02x",stonesoup_heap_buff_64[stonesoup_ss_i]);
        }
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: AFTER");
        tracepoint(stonesoup_trace, variable_buffer, "stonesoup_heap_buff_64",
stonesoup_heap_buff_64, "CROSSOVER-STATE");
        tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
        stonesoup_printf("\n");
        free( stonesoup_heap_buff_64);
```

```
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, variable_signed_integral, "stonesoup_ss_i",
stonesoup_ss_i, &stonesoup_ss_i, "FINAL-STATE");
        tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
    }
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 *****************************************************************************/
int main(int argc,char *argv[])
{
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    char* tainted_buff = argv[1];
    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
    return 0;
}
```

```
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-126C Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1
#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1
#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1
#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1
#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1
#endif /* NOTHING DEFINED */

#if STONESOUP_SNIPPET_INCLUDES
/*****************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 *****************************************************************************/
#include <stdlib.h> /* malloc */
#include <stdio.h> /* fprintf */
#include <string.h> /* memset */
#include <errno.h> /* errno */
/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_INCLUDES */
#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 *****************************************************************************/
#include "output-handlers/handler.h"
```

```
#endif /* STONESOUP_SNIPPET_DEBUG */
struct stonesoup_struct {
    char before[64];
    char buffer[64];
    char after[64];
};
#endif /* STONESOUP_SNIPPET_SUPPORT */

#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 *****************************************************************************/
void weakness(char * stonesoup_tainted_buff) {
#define STONESOUP_TAINT_SOURCE stonesoup_tainted_buff
#endif /* STONESOUP_SNIPPET_DEBUG */
#if STONESOUP_SNIPPET_BODY_DECLARATIONS
    int stonesoup_i = 0;
    int stonesoup_buff_size = 0;

    struct stonesoup_struct* stonesoup_data = NULL;
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_start, "CWE126", "C", "Buffer Over-read");
    tracepoint(stonesoup_trace, variable_signed_integral, "stonesoup_i", stonesoup_i,
&stonesoup_i, "INITIAL-STATE");
    tracepoint(stonesoup_trace, variable_signed_integral, "stonesoup_buff_size",
stonesoup_buff_size, &stonesoup_buff_size, "INITIAL-STATE");
#endif /* STONESOUP_TRACE */

    stonesoup_data = (struct stonesoup_struct*) malloc(sizeof(struct
stonesoup_struct));
    if (stonesoup_data != NULL) {
        memset(stonesoup_data->before, 'A', 63);
        stonesoup_data->before[63] = '\0';

        memset(stonesoup_data->buffer, 'Q', 63);
        stonesoup_data->buffer[63] = '\0';

        memset(stonesoup_data->after, 'A', 63);
        stonesoup_data->after[63] = '\0';
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, variable_buffer, "stonesoup_data->before",
stonesoup_data->before, "INITIAL-STATE");
        tracepoint(stonesoup_trace, variable_buffer, "stonesoup_data->buffer",
stonesoup_data->buffer, "INITIAL-STATE");
        tracepoint(stonesoup_trace, variable_buffer, "stonesoup_data->after",
stonesoup_data->after, "INITIAL-STATE");
        tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
        tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
        stonesoup_buff_size = ((int )(strlen(STONESOUP_TAINT_SOURCE)));
```

```
        memcpy(stonesoup_data->buffer, STONESOUP_TAINT_SOURCE, 64);
        for (; stonesoup_i < stonesoup_buff_size; ++stonesoup_i){
            /* STONESOUP: CROSSOVER-POINT (Buffer Overread) */
            /* STONESOUP: TRIGGER-POINT (Buffer Overread) */
            stonesoup_printf("%x",stonesoup_data->buffer[stonesoup_i]);
        }
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: AFTER");
        tracepoint(stonesoup_trace, variable_buffer, "stonesoup_data->buffer",
stonesoup_data->buffer, "CROSSOVER-STATE");
        tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
        stonesoup_printf("\n");
        free( stonesoup_data);
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, variable_signed_integral, "stonesoup_i",
stonesoup_i, &stonesoup_i, "FINAL-STATE");
        tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
    }
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_DEBUG
/******************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 ******************************************************************************/
int main(int argc,char *argv[])
{
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    char* tainted_buff = argv[1];
    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
    return 0;
```

```
}
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-126D Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1
#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1
#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1
#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1
#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1
#endif /* NOTHING DEFINED */

#if STONESOUP_SNIPPET_INCLUDES
/*****************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 *****************************************************************************/
#include <stdlib.h> /* malloc */
#include <stdio.h> /* fprintf */
#include <string.h> /* memset */
#include <errno.h> /* errno */
/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_INCLUDES */
#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 *****************************************************************************/
```

```
#include "output-handlers/handler.h"
#endif /* STONESOUP_SNIPPET_DEBUG */
struct stonesoup_struct {
    char before[64];
    char buffer[64];
    char after[64];
};
#endif /* STONESOUP_SNIPPET_SUPPORT */

#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 *****************************************************************************/
void weakness(char * stonesoup_tainted_buff) {
#define STONESOUP_TAINT_SOURCE stonesoup_tainted_buff
#endif /* STONESOUP_SNIPPET_DEBUG */
#if STONESOUP_SNIPPET_BODY_DECLARATIONS
    int stonesoup_i = 0;
    int stonesoup_buff_size = 0;

    struct stonesoup_struct stonesoup_data;
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_start, "CWE126", "D", "Buffer Over-read");
#endif /* STONESOUP_TRACE */

    for (stonesoup_i = 0; stonesoup_i < 63; stonesoup_i++) {
        stonesoup_data.before[stonesoup_i] = 'A';
    }
    stonesoup_data.before[63] = '\0';

    for (stonesoup_i = 0; stonesoup_i < 63; stonesoup_i++) {
        stonesoup_data.buffer[stonesoup_i] = 'Q';
    }
    stonesoup_data.buffer[63] = '\0';

    for (stonesoup_i = 0; stonesoup_i < 63; stonesoup_i++) {
        stonesoup_data.after[stonesoup_i] = 'A';
    }
    stonesoup_data.after[63] = '\0';

#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, variable_buffer, "stonesoup_data.before",
stonesoup_data.before, "INITIAL-STATE");
    tracepoint(stonesoup_trace, variable_buffer, "stonesoup_data.buffer",
stonesoup_data.buffer, "INITIAL-STATE");
    tracepoint(stonesoup_trace, variable_buffer, "stonesoup_data.after",
stonesoup_data.after, "INITIAL-STATE");
    tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
    tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
```

```
    stonesoup_buff_size = ((int )(strlen(STONESOUP_TAINT_SOURCE)));
    memcpy(stonesoup_data.buffer, STONESOUP_TAINT_SOURCE, 64);
    for (; stonesoup_i < stonesoup_buff_size; ++stonesoup_i){
        /* STONESOUP: CROSSOVER-POINT (Buffer Overread) */
        /* STONESOUP: TRIGGER-POINT (Buffer Overread) */
        stonesoup_printf("%x",stonesoup_data.buffer[stonesoup_i]);
    }
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: AFTER");
    tracepoint(stonesoup_trace, variable_buffer, "stonesoup_data.buffer",
stonesoup_data.buffer, "CROSSOVER-STATE");
    tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
    stonesoup_printf("\n");
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, variable_signed_integral, "stonesoup_i", stonesoup_i,
&stonesoup_i, "FINAL-STATE");
    tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 ******************************************************************************/
int main(int argc,char *argv[])
{
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    char* tainted_buff = argv[1];
    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
    return 0;
```

```
}
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-127A Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1
#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1
#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1
#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1
#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1
#endif /* NOTHING DEFINED */

#if STONESOUP_SNIPPET_INCLUDES
/*****************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 *****************************************************************************/
#include <stdlib.h> /* malloc */
#include <stdio.h> /* fprintf */
#include <string.h> /* memset */
#include <errno.h> /* errno */
/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_INCLUDES */
#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 *****************************************************************************/
```

```
#include "output-handlers/handler.h"
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_SUPPORT */

#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 *****************************************************************************/
void weakness(char * stonesoup_tainted_buff) {
#define STONESOUP_TAINT_SOURCE stonesoup_tainted_buff
#endif /* STONESOUP_SNIPPET_DEBUG */
#define STONESOUP_EXPRESSION_1 "INPUT_STRING"
#if STONESOUP_SNIPPET_BODY_DECLARATIONS
 signed char *stonesoup_input_string = 0;
 int stonesoup_stack_buff[128];
 int stonesoup_other_buff[200];
 int stonesoup_ss_i = 0;
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_start, "CWE127", "A", "Buffer Under-read");
#endif /* STONESOUP_TRACE */
 stonesoup_input_string = (signed char *) getenv(STONESOUP_EXPRESSION_1);
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, variable_buffer, "((char *)stonesoup_input_string)",
((char *)stonesoup_input_string), "INITIAL-STATE");
    tracepoint(stonesoup_trace, variable_address, "stonesoup_stack_buff",
stonesoup_stack_buff, "INITIAL-STATE");
    tracepoint(stonesoup_trace, variable_address, "stonesoup_other_buff",
stonesoup_other_buff, "INITIAL-STATE");
#endif /* STONESOUP_TRACE */
    if (stonesoup_input_string != 0) {
        memset(stonesoup_stack_buff,0,sizeof(stonesoup_stack_buff));
        for (stonesoup_ss_i = 0; stonesoup_ss_i < 200; ++stonesoup_ss_i) {
   stonesoup_other_buff[stonesoup_ss_i] = 5555;
        }
        for (stonesoup_ss_i = 0; stonesoup_ss_i < strlen((char
*)stonesoup_input_string); ++stonesoup_ss_i) {
   if (stonesoup_input_string[stonesoup_ss_i] < 0)
    continue;
   ++stonesoup_stack_buff[stonesoup_input_string[stonesoup_ss_i]];
        }
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
        tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
        for (stonesoup_ss_i = 0; stonesoup_ss_i < strlen(STONESOUP_TAINT_SOURCE);
++stonesoup_ss_i) {
   /* STONESOUP: CROSSOVER-POINT (Buffer Underread) */
   /* STONESOUP: TRIGGER-POINT (Buffer Underread) */
#if STONESOUP_TRACE
            tracepoint(stonesoup_trace, variable_signed_integral,
```

```
"((int)STONESOUP_TAINT_SOURCE[stonesoup_ss_i])",
((int)STONESOUP_TAINT_SOURCE[stonesoup_ss_i]),
&(STONESOUP_TAINT_SOURCE[stonesoup_ss_i]), "TRIGGER-STATE");
#endif /* STONESOUP_TRACE */
   stonesoup_printf("value %c appears: %d times\n",
       STONESOUP_TAINT_SOURCE[stonesoup_ss_i],
       stonesoup_stack_buff[(int) STONESOUP_TAINT_SOURCE[stonesoup_ss_i]]);
        }
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: AFTER");
        tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
    }
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 ******************************************************************************/
int main(int argc,char *argv[])
{
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    char* tainted_buff = argv[1];

    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
    return 0;
}
```

```
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-127B Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1
#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1
#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1
#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1
#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1
#endif /* NOTHING DEFINED */

#if STONESOUP_SNIPPET_INCLUDES
/*****************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 *****************************************************************************/
#include <stdlib.h> /* malloc */
#include <stdio.h> /* fprintf */
#include <string.h> /* memset */
#include <errno.h> /* errno */
/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_INCLUDES */
#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 *****************************************************************************/
#include "output-handlers/handler.h"
```

```
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_SUPPORT */

#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 *****************************************************************************/
void weakness(char * stonesoup_tainted_buff) {
#define STONESOUP_TAINT_SOURCE stonesoup_tainted_buff
#endif /* STONESOUP_SNIPPET_DEBUG */
#if STONESOUP_SNIPPET_BODY_DECLARATIONS
    int stonesoup_i = 0;
    char * stonesoup_other_buff = 0;
    int stonesoup_buff_size = 0;
    int stonesoup_other_size = 0;
    int stonesoup_size;
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_start, "CWE127", "B", "Buffer Under-read");
#endif /* STONESOUP_TRACE */
    stonesoup_buff_size = strlen(STONESOUP_TAINT_SOURCE) + 1;

    stonesoup_other_size = 64;
    stonesoup_other_buff = (char*) malloc (stonesoup_other_size * sizeof (char));
    if (stonesoup_other_buff != NULL) {
        stonesoup_size = stonesoup_other_size < stonesoup_buff_size ?
stonesoup_other_size : stonesoup_buff_size;
        for (stonesoup_i = 0; stonesoup_i < stonesoup_size; stonesoup_i++) {
            stonesoup_other_buff[stonesoup_other_size - stonesoup_i - 1] =
                STONESOUP_TAINT_SOURCE[stonesoup_buff_size - stonesoup_i - 1];
        }

#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
        tracepoint(stonesoup_trace, trace_point, "AFTER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
        for (stonesoup_i = 0; stonesoup_i < stonesoup_buff_size; stonesoup_i++) {
                /* STONESOUP: CROSSOVER-POINT (Buffer Underread) */
                /* STONESOUP: TRIGGER-POINT (Buffer Underread) */
                stonesoup_printf("%02x",stonesoup_other_buff[stonesoup_other_size -
stonesoup_i - 1]);
        }
        stonesoup_printf("\n");
        free (stonesoup_other_buff);
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, variable_signed_integral, "stonesoup_other_size",
stonesoup_other_size, &stonesoup_other_size, "TRIGGER-STATE");
        tracepoint(stonesoup_trace, variable_signed_integral, "stonesoup_buff_size",
stonesoup_buff_size, &stonesoup_buff_size, "TRIGGER-STATE");
        tracepoint(stonesoup_trace, trace_point, "AFTER-POINT: AFTER");
        tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
```

Approved for public release; distribution unlimited.

```
#endif /* STONESOUP_TRACE */
    }
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 *****************************************************************************/
int main(int argc,char *argv[])
{
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    char* tainted_buff = argv[1];

    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
    return 0;
}
```

Approved for public release; distribution unlimited.

```
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-127C Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1
#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1
#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1
#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1
#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1
#endif /* NOTHING DEFINED */

#if STONESOUP_SNIPPET_INCLUDES
/*****************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 *****************************************************************************/
#include <stdlib.h> /* malloc */
#include <stdio.h> /* fprintf */
#include <string.h> /* memset */
#include <errno.h> /* errno */
/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_INCLUDES */
#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 *****************************************************************************/
#include "output-handlers/handler.h"
```

```
#endif /* STONESOUP_SNIPPET_DEBUG */
struct stonesoup_struct {
    int before[200];
    int buffer[128];
    int after[200];
};
#endif /* STONESOUP_SNIPPET_SUPPORT */

#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 *****************************************************************************/
void weakness(char * stonesoup_tainted_buff) {
#define STONESOUP_TAINT_SOURCE stonesoup_tainted_buff
#endif /* STONESOUP_SNIPPET_DEBUG */
#define STONESOUP_EXPRESSION_1 "INPUT_STRING"
#if STONESOUP_SNIPPET_BODY_DECLARATIONS
    signed char *stonesoup_input_string = 0;
    struct stonesoup_struct * stonesoup_data = 0;
    int stonesoup_i = 0;
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_start, "CWE127", "C", "Buffer Under-read");
#endif /* STONESOUP_TRACE */
    stonesoup_input_string = (signed char *) getenv(STONESOUP_EXPRESSION_1);
    stonesoup_data = (struct stonesoup_struct *) malloc (sizeof (struct
stonesoup_struct));
    if (stonesoup_data != NULL) {
        if (stonesoup_input_string != 0) {
            memset(stonesoup_data->buffer, 0, 128);
            for (stonesoup_i = 0; stonesoup_i < 200; ++stonesoup_i) {
                stonesoup_data->before[stonesoup_i] = 5555;
                stonesoup_data->after[stonesoup_i] = 5555;
            }

            for (stonesoup_i = 0; stonesoup_i < strlen((char *)
stonesoup_input_string); ++stonesoup_i) {
                if (stonesoup_input_string[stonesoup_i] < 0)
                    continue;
                ++stonesoup_data->buffer[stonesoup_input_string[stonesoup_i]];
            }

#if STONESOUP_TRACE
            tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
            tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
            for (stonesoup_i = 0; stonesoup_i < strlen(STONESOUP_TAINT_SOURCE);
++stonesoup_i) {
                /* STONESOUP: CROSSOVER-POINT (Buffer Underread) */
                /* STONESOUP: TRIGGER-POINT (Buffer Underread) */
                stonesoup_printf("value %c appears: %d times\n",
```

```
                    STONESOUP_TAINT_SOURCE[stonesoup_i],
                    stonesoup_data->buffer[(int)
STONESOUP_TAINT_SOURCE[stonesoup_i]]);
            }
#if STONESOUP_TRACE
            tracepoint(stonesoup_trace, variable_signed_integral, "((int)
STONESOUP_TAINT_SOURCE[stonesoup_i])", ((int) STONESOUP_TAINT_SOURCE[stonesoup_i]),
&(STONESOUP_TAINT_SOURCE[stonesoup_i]), "TRIGGER-STATE");
            tracepoint(stonesoup_trace, trace_point, "AFTER-POINT: AFTER");
            tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
        }
        free (stonesoup_data);
    }
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 *******************************************************************************/
int main(int argc,char *argv[])
{
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    char* tainted_buff = argv[1];

    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
    return 0;
}
```

```
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-127D Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1
#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1
#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1
#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1
#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1
#endif /* NOTHING DEFINED */

#if STONESOUP_SNIPPET_INCLUDES
/*******************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 *******************************************************************************/
#include <stdlib.h> /* malloc */
#include <stdio.h> /* fprintf */
#include <string.h> /* memset */
#include <errno.h> /* errno */
/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_INCLUDES */
#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 *******************************************************************************/
#include "output-handlers/handler.h"
```

```
#endif /* STONESOUP_SNIPPET_DEBUG */
struct stonesoup_struct {
    int before[200];
    int buffer[128];
    int after[200];
};
#endif /* STONESOUP_SNIPPET_SUPPORT */

#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 ******************************************************************************/
void weakness(char * stonesoup_tainted_buff) {
#define STONESOUP_TAINT_SOURCE stonesoup_tainted_buff
#endif /* STONESOUP_SNIPPET_DEBUG */
#define STONESOUP_EXPRESSION_1 "INPUT_STRING"
#if STONESOUP_SNIPPET_BODY_DECLARATIONS
    signed char *stonesoup_input_string = 0;
    struct stonesoup_struct stonesoup_data;
    int stonesoup_i = 0;
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_start, "CWE127", "D", "Buffer Under-read");
#endif /* STONESOUP_TRACE */
    stonesoup_input_string = (signed char *) getenv(STONESOUP_EXPRESSION_1);
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, variable_buffer, "STONESOUP_TAINT_SOURCE",
STONESOUP_TAINT_SOURCE, "INITIAL-STATE");
    tracepoint(stonesoup_trace, variable_buffer, "stonesoup_input_string",
stonesoup_input_string, "INITIAL-STATE");
#endif /* STONESOUP_TRACE */
    if (stonesoup_input_string != 0) {
        for (stonesoup_i = 0; stonesoup_i < 128; ++stonesoup_i) {
            stonesoup_data.buffer[stonesoup_i] = 0;
        }
        for (stonesoup_i = 0; stonesoup_i < 200; ++stonesoup_i) {
            stonesoup_data.before[stonesoup_i] = 5555;
            stonesoup_data.after[stonesoup_i] = 5555;
        }

        for (stonesoup_i = 0; stonesoup_i < strlen((char *) stonesoup_input_string);
++stonesoup_i) {
            if (stonesoup_input_string[stonesoup_i] < 0)
                continue;
            ++stonesoup_data.buffer[stonesoup_input_string[stonesoup_i]];
        }

#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
        tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
```

```
        for (stonesoup_i = 0; stonesoup_i < strlen(STONESOUP_TAINT_SOURCE);
++stonesoup_i) {
            /* STONESOUP: CROSSOVER-POINT (Buffer Underread) */
            /* STONESOUP: TRIGGER-POINT (Buffer Underread) */
            stonesoup_printf("value %c appears: %d times\n",
                STONESOUP_TAINT_SOURCE[stonesoup_i],
                stonesoup_data.buffer[(int) STONESOUP_TAINT_SOURCE[stonesoup_i]]);
        }
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, variable_signed_integral, "((int)
STONESOUP_TAINT_SOURCE[stonesoup_i])", ((int) STONESOUP_TAINT_SOURCE[stonesoup_i]),
&(STONESOUP_TAINT_SOURCE[stonesoup_i]), "TRIGGER-STATE");
        tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: AFTER");
        tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
    }
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_DEBUG
/******************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 ******************************************************************************/
int main(int argc,char *argv[])
{
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    char* tainted_buff = argv[1];

    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
    return 0;
}
```

```
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-129A Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
    (!STONESOUP_SNIPPET_INCLUDES && \
     !STONESOUP_SNIPPET_SUPPORT && \
     !STONESOUP_SNIPPET_BODY_STATEMENTS && \
     !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1
#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1
#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1
#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1
#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1
#endif /* NOTHING DEFINED */

#if STONESOUP_SNIPPET_INCLUDES
/*****************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 *****************************************************************************/
#include <stdlib.h> /* malloc */
#include <stdio.h> /* fprintf */
#include <string.h> /* memset */
#include <errno.h> /* errno */
/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_INCLUDES */
#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 *****************************************************************************/
#include "output-handlers/handler.h"
```

```
#endif /* STONESOUP_SNIPPET_DEBUG */
int stonesoup_single_global_variable = 0;
int stonesoup_canary_function_1()
{
#if STONESOUP_TRACE
  tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE,
"stonesoup_canary_function_1");
#endif /* STONESOUP_TRACE */
  ++stonesoup_single_global_variable;
  return 1;
}
int stonesoup_canary_function_2()
{
#if STONESOUP_TRACE
  tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE,
"stonesoup_canary_function_2");
#endif /* STONESOUP_TRACE */
  ++stonesoup_single_global_variable;
  return 2;
}
#endif /* STONESOUP_SNIPPET_SUPPORT */

#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 *****************************************************************************/
void weakness(char * stonesoup_tainted_buff) {
#define STONESOUP_TAINT_SOURCE stonesoup_tainted_buff
#endif /* STONESOUP_SNIPPET_DEBUG */
#if STONESOUP_SNIPPET_BODY_DECLARATIONS
    int stonesoup_i = 0;
    int stonesoup_index;
    int (*stonesoup_after_ptr[1])();
    unsigned char stonesoup_count[62];
    int (*stonesoup_before_ptr[1])();
    char stonesoup_str_buf[40] = {0};
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_start, "CWE129", "A", "Improper Validation of
Array Index");
#endif /* STONESOUP_TRACE */
    strncpy(stonesoup_str_buf,STONESOUP_TAINT_SOURCE,39);
    stonesoup_str_buf[39] = 0;
    for (stonesoup_i = 0; stonesoup_i < 62; stonesoup_i++) {
        stonesoup_count[stonesoup_i] = 0;
    }
    if (strlen(stonesoup_str_buf) > 1 && stonesoup_str_buf[0] > 'a') {
        stonesoup_before_ptr[0] = stonesoup_canary_function_1;
        stonesoup_after_ptr[0] = stonesoup_canary_function_1;
    }
    else {
```

```
        stonesoup_before_ptr[0] = stonesoup_canary_function_2;
        stonesoup_after_ptr[0] = stonesoup_canary_function_2;
    }
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
    tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
    for (stonesoup_i = 0; stonesoup_i < strlen(stonesoup_str_buf); stonesoup_i++)
        /* STONESOUP: CROSSOVER-POINT (Improper Validation of Array Index) */
        /* STONESOUP: TRIGGER-POINT (Improper Validation of Array Index: Ascii Bounds)
*/
    {
        if (stonesoup_str_buf[stonesoup_i] > 96) {
            stonesoup_index = stonesoup_str_buf[stonesoup_i] - 'a' + 36;
            if (stonesoup_count[stonesoup_index] < 255)
                stonesoup_count[stonesoup_index]++;
        }
        else if (stonesoup_str_buf[stonesoup_i] < 58) {
            stonesoup_index = stonesoup_str_buf[stonesoup_i] - 48;
#if STONESOUP_TRACE
            tracepoint(stonesoup_trace, variable_signed_integral, "stonesoup_index",
stonesoup_index, &stonesoup_index, "TRIGGER-POINT");
#endif /* STONESOUP_TRACE */
            if (stonesoup_count[stonesoup_index] < 255)
                stonesoup_count[stonesoup_index]++;
        }
        else {
            stonesoup_index = stonesoup_str_buf[stonesoup_i] - 'A' + 10;
            if (stonesoup_count[stonesoup_index] < 255)
                stonesoup_count[stonesoup_index]++;
        }
    }
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: AFTER");
    tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
    for (stonesoup_i = 0; stonesoup_i < 62; stonesoup_i++) {
        stonesoup_printf("index %d: %d\n",stonesoup_i,stonesoup_count[stonesoup_i]);
    }
    stonesoup_printf("%d\n",stonesoup_before_ptr[0]());
    stonesoup_printf("%d\n",stonesoup_after_ptr[0]());
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 *****************************************************************************/
int main(int argc,char *argv[])
{
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
```

```
        exit(1);
    }
    char* tainted_buff = argv[1];
    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
    return 0;
}
```

Approved for public release; distribution unlimited.

```
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-129B Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
    (!STONESOUP_SNIPPET_INCLUDES && \
     !STONESOUP_SNIPPET_SUPPORT && \
     !STONESOUP_SNIPPET_BODY_STATEMENTS && \
     !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1
#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1
#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1
#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1
#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1
#endif /* NOTHING DEFINED */

#if STONESOUP_SNIPPET_INCLUDES
/*****************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 *****************************************************************************/
#include <stdlib.h> /* malloc */
#include <stdio.h> /* fprintf */
#include <string.h> /* memset */
#include <errno.h> /* errno */
/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_INCLUDES */
#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 *****************************************************************************/
#include "output-handlers/handler.h"
```

```
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_SUPPORT */

#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 ******************************************************************************/
void weakness(char * stonesoup_tainted_buff) {
#define STONESOUP_TAINT_SOURCE stonesoup_tainted_buff
#endif /* STONESOUP_SNIPPET_DEBUG */
#if STONESOUP_SNIPPET_BODY_DECLARATIONS
    int stonesoup_i;
    int stonesoup_index;
    char **stonesoup_ptr_after = 0;
    unsigned char *stonesoup_count = 0;
    char **stonesoup_ptr_before = 0;
    unsigned char stonesoup_str_buf[40] = {0};
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_start, "CWE129", "B", "Improper Validation of
Array Index");
#endif /* STONESOUP_TRACE */
    stonesoup_ptr_after = malloc(sizeof(char *));
    if (stonesoup_ptr_after == 0) {
        stonesoup_printf("Error: Failed to allocate memory\n");
        exit(1);
    }
    stonesoup_count = malloc(sizeof(unsigned char ) * 62);
    if (stonesoup_count == 0) {
        stonesoup_printf("Error: Failed to allocate memory\n");
        exit(1);
    }
    stonesoup_ptr_before = malloc(sizeof(char *));
    if (stonesoup_ptr_before == 0) {
        stonesoup_printf("Error: Failed to allocate memory\n");
        exit(1);
    }
    *stonesoup_ptr_before = STONESOUP_TAINT_SOURCE;
    *stonesoup_ptr_after = STONESOUP_TAINT_SOURCE;
    strncpy((char*) stonesoup_str_buf,STONESOUP_TAINT_SOURCE,39);
    stonesoup_str_buf[39] = 0;
    for (stonesoup_i = 0; stonesoup_i < 62; stonesoup_i++) {
        stonesoup_count[stonesoup_i] = 0;
    }

#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
    tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
    for (stonesoup_i = 0; stonesoup_i < strlen(STONESOUP_TAINT_SOURCE); stonesoup_i++)
{
```

```
        /* STONESOUP: CROSSOVER-POINT (Improper Validation of Array Index) */
        /* STONESOUP: TRIGGER-POINT (Improper Validation of Array Index: Ascii Bounds)
*/
        if (STONESOUP_TAINT_SOURCE[stonesoup_i] > 96) {
            stonesoup_index = STONESOUP_TAINT_SOURCE[stonesoup_i] - 'a' + 36;
            if (stonesoup_count[stonesoup_index] < 255)
                stonesoup_count[stonesoup_index]++;
        }
        else if (STONESOUP_TAINT_SOURCE[stonesoup_i] < 58) {
            stonesoup_index = STONESOUP_TAINT_SOURCE[stonesoup_i] - 48;
#if STONESOUP_TRACE
            tracepoint(stonesoup_trace, variable_signed_integral, "stonesoup_index",
stonesoup_index, &stonesoup_index, "TRIGGER-STATE");
#endif /* STONESOUP_TRACE */
            if (stonesoup_count[stonesoup_index] < 255)
                stonesoup_count[stonesoup_index]++;
        }
        else {
            stonesoup_index = STONESOUP_TAINT_SOURCE[stonesoup_i] - 'A' + 10;
            if (stonesoup_count[stonesoup_index] < 255)
                stonesoup_count[stonesoup_index]++;
        }
    }
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: AFTER");
    tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
    for (stonesoup_i = 0; stonesoup_i < 62; stonesoup_i++)
        stonesoup_printf("index %d: %d\n",stonesoup_i,stonesoup_count[stonesoup_i]);
    stonesoup_printf("%d %d\n",strlen( *stonesoup_ptr_before),strlen(
*stonesoup_ptr_after));
    if (stonesoup_ptr_before != 0) {
        free(stonesoup_ptr_before);
    }
    if (stonesoup_count != 0) {
        free(stonesoup_count);
    }
    if (stonesoup_ptr_after != 0) {
        free(stonesoup_ptr_after);
    }
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 ******************************************************************************/
int main(int argc,char *argv[])
{
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
```

```
    }
    char* tainted_buff = argv[1];
    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
    return 0;
}
```

```
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-134A Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1
#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1
#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1
#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1
#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1
#endif /* NOTHING DEFINED */

#if STONESOUP_SNIPPET_INCLUDES
/*****************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 *****************************************************************************/
#include <stdlib.h> /* malloc */
#include <stdio.h> /* fprintf */
#include <string.h> /* memset */
#include <errno.h> /* errno */
/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_INCLUDES */
#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 *****************************************************************************/
#include "output-handlers/handler.h"
```

```
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_SUPPORT */

#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 *****************************************************************************/
void weakness(char * stonesoup_tainted_buff) {
#define STONESOUP_TAINT_SOURCE stonesoup_tainted_buff
#endif /* STONESOUP_SNIPPET_DEBUG */
#if STONESOUP_SNIPPET_BODY_DECLARATIONS
 char stonesoup_buffer_stack[128] = {0};
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
 tracepoint(stonesoup_trace, weakness_start, "CWE134", "A", "Uncontrolled Format
String");
    tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
    tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
    /* STONESOUP: CROSSOVER-POINT (Uncontrolled Format String) */
 /* STONESOUP: TRIGGER-POINT (Uncontrolled Format String) */
    sprintf(stonesoup_buffer_stack,STONESOUP_TAINT_SOURCE);
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, variable_buffer, "stonesoup_buffer_stack",
stonesoup_buffer_stack, "TRIGGER-STATE");
#endif /* STONESOUP_TRACE */
    stonesoup_printf("%s\n",stonesoup_buffer_stack);
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: AFTER");
    tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 *****************************************************************************/
int main(int argc,char *argv[])
{
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    char* tainted_buff = argv[1];

    if (tainted_buff != NULL) {
```

```
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
    return 0;
}
```

```
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-134B Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1
#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1
#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1
#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1
#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1
#endif /* NOTHING DEFINED */

#if STONESOUP_SNIPPET_INCLUDES
/*****************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 *****************************************************************************/
#include <stdlib.h> /* malloc */
#include <stdio.h> /* fprintf */
#include <string.h> /* memset */
#include <errno.h> /* errno */
/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_INCLUDES */
#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 *****************************************************************************/
#include "output-handlers/handler.h"
```

```
#endif /* STONESOUP_SNIPPET_DEBUG */
int stonesoup_toupper(int c)
{
  if (c >= 97 && c <= 122) {
    return c - 32;
  }
  return c;
}
#endif /* STONESOUP_SNIPPET_SUPPORT */

#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 *****************************************************************************/
void weakness(char * stonesoup_tainted_buff) {
#define STONESOUP_TAINT_SOURCE stonesoup_tainted_buff
#endif /* STONESOUP_SNIPPET_DEBUG */
#if STONESOUP_SNIPPET_BODY_DECLARATIONS
 int stonesoup_oc_i = 0;
 int stonesoup_len;
 char *stonesoup_buffer_heap = 0;
 char **stonesoup_buf_ptr = 0;
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
 tracepoint(stonesoup_trace, weakness_start, "CWE134", "B", "Uncontrolled Format
String");
#endif /* STONESOUP_TRACE */
    stonesoup_buffer_heap = malloc(sizeof(char ) * 128);
    if (stonesoup_buffer_heap == 0) {
        stonesoup_printf("Error: Failed to allocate memory\n");
        exit(1);
    }
    memset(stonesoup_buffer_heap,0,128);
    stonesoup_buf_ptr = malloc(sizeof(char *));
    if (stonesoup_buf_ptr == 0) {
        stonesoup_printf("Error: Failed to allocate memory\n");
        exit(1);
    }
    *stonesoup_buf_ptr = stonesoup_buffer_heap;
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
    tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
 /* STONESOUP: CROSSOVER-POINT (Uncontrolled Format String) */
 /* STONESOUP: TRIGGER-POINT (Uncontrolled Format String) */
    sprintf(stonesoup_buffer_heap,STONESOUP_TAINT_SOURCE);
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, variable_buffer, "stonesoup_buffer_heap",
stonesoup_buffer_heap, "TRIGGER-STATE");
#endif /* STONESOUP_TRACE */
    stonesoup_printf("%s\n",stonesoup_buffer_heap);
```

Approved for public release; distribution unlimited.

```
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: AFTER");
    tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
    stonesoup_len = strlen( *stonesoup_buf_ptr);
    for (; stonesoup_oc_i < stonesoup_len; ++stonesoup_oc_i) {
        stonesoup_buffer_heap[stonesoup_oc_i] =
stonesoup_toupper(stonesoup_buffer_heap[stonesoup_oc_i]);
    }
    stonesoup_printf("%s\n",stonesoup_buffer_heap);
    if (stonesoup_buffer_heap != 0) {
        free(stonesoup_buffer_heap);
    }
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 *******************************************************************************/
int main(int argc,char *argv[])
{
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    char* tainted_buff = argv[1];

    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
    return 0;
```

Approved for public release; distribution unlimited.

```
}
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-170A Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1
#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1
#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1
#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1
#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1
#endif /* NOTHING DEFINED */

#if STONESOUP_SNIPPET_INCLUDES
/*******************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 *******************************************************************************/
#include <stdlib.h> /* malloc */
#include <stdio.h> /* fprintf */
#include <string.h> /* memset */
#include <errno.h> /* errno */
#include <unistd.h> /* read */
#include <fcntl.h> /* open */
/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_INCLUDES */
#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
```

```
 ******************************************************************************/
#include "output-handlers/handler.h"
#endif /* STONESOUP_SNIPPET_DEBUG */
int stonesoup_toupper(int c)
{
  if (c >= 97 && c <= 122) {
    return c - 32;
  }
  return c;
}
#endif /* STONESOUP_SNIPPET_SUPPORT */

#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 ******************************************************************************/
void weakness(char * stonesoup_tainted_buff) {
#define STONESOUP_TAINT_SOURCE stonesoup_tainted_buff
#endif /* STONESOUP_SNIPPET_DEBUG */
#if STONESOUP_SNIPPET_BODY_DECLARATIONS
 int stonesoup_oc_i = 0;
 int stonesoup_file_desc;
 char stonesoup_buffer[128];
 char stonesoup_input_buf[128] = {0};
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
 tracepoint(stonesoup_trace, weakness_start, "CWE170", "A", "Improper Null
Termination");
#endif /* STONESOUP_TRACE */
    memset(stonesoup_buffer,'x',128);
    stonesoup_buffer[127] = 0;
    stonesoup_file_desc = open(STONESOUP_TAINT_SOURCE,0);
    if (stonesoup_file_desc > -1) {
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
  /* STONESOUP: CROSSOVER-POINT (Improper Null Termination) */
        read(stonesoup_file_desc,stonesoup_input_buf,128);
        close(stonesoup_file_desc);
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
        tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
  /* STONESOUP: TRIGGER-POINT (Improper Null Termination: Stack Over Read) */
        strcpy(stonesoup_buffer,stonesoup_input_buf);
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, variable_signed_integral,
"(stonesoup_buffer[127])", (stonesoup_buffer[127]), &(stonesoup_buffer[127]),
"TRIGGER-STATE");
#endif /* STONESOUP_TRACE */
        for (; stonesoup_oc_i < strlen(stonesoup_buffer); ++stonesoup_oc_i) {
```

```
   stonesoup_buffer[stonesoup_oc_i] =
stonesoup_toupper(stonesoup_buffer[stonesoup_oc_i]);
        }
        stonesoup_printf("%s\n",stonesoup_buffer);
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
    }
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 *****************************************************************************/
int main(int argc,char *argv[])
{
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    char* tainted_buff = argv[1];

    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
    return 0;
```

```
}
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-170B Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1
#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1
#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1
#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1
#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1
#endif /* NOTHING DEFINED */

#if STONESOUP_SNIPPET_INCLUDES
/*****************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 *****************************************************************************/
#include <stdlib.h> /* malloc */
#include <stdio.h> /* fprintf */
#include <string.h> /* memset */
#include <errno.h> /* errno */
#include <unistd.h> /* read */
#include <fcntl.h> /* open */
/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_INCLUDES */
#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
```

```
 ******************************************************************************/
#include "output-handlers/handler.h"
#endif /* STONESOUP_SNIPPET_DEBUG */
int stonesoup_toupper(int c)
{
  if (c >= 97 && c <= 122) {
    return c - 32;
  }
  return c;
}
#endif /* STONESOUP_SNIPPET_SUPPORT */

#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 ******************************************************************************/
void weakness(char * stonesoup_tainted_buff) {
#define STONESOUP_TAINT_SOURCE stonesoup_tainted_buff
#endif /* STONESOUP_SNIPPET_DEBUG */
#if STONESOUP_SNIPPET_BODY_DECLARATIONS
 int stonesoup_oc_i = 0;
 int stonesoup_buf_ptr_len;
 char *stonesoup_new_input = "new test input";
 const int stonesoup_MAXLEN = 16;
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_start, "CWE170", "B", "Improper Null
Termination");
#endif /* STONESOUP_TRACE */
 int stonesoup_file_desc;
 char stonesoup_input_buf[stonesoup_MAXLEN];
 char **stonesoup_buf_ptr = 0;
 char *stonesoup_path_buf = 0;
 stonesoup_path_buf = malloc(sizeof(char ) * 64);
    if (stonesoup_path_buf == 0) {
        stonesoup_printf("Error: Failed to allocate memory\n");
        exit(1);
    }
    stonesoup_buf_ptr = malloc(sizeof(char *));
    if (stonesoup_buf_ptr == 0) {
        stonesoup_printf("Error: Failed to allocate memory\n");
        exit(1);
    }
    *stonesoup_buf_ptr = stonesoup_path_buf;
    memset(stonesoup_input_buf,0,16);
    memset(stonesoup_path_buf,'a',64);
    stonesoup_path_buf[63] = 0;
    stonesoup_file_desc = open(STONESOUP_TAINT_SOURCE,0);
    if (stonesoup_file_desc > -1) {
        read(stonesoup_file_desc,stonesoup_input_buf,stonesoup_MAXLEN);
        close(stonesoup_file_desc);
```

```
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
  /* STONESOUP: CROSSOVER-POINT (Improper Null Termination) */
        strncpy(stonesoup_path_buf,stonesoup_input_buf,stonesoup_MAXLEN);
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
        tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
  /* STONESOUP: TRIGGER-POINT (Improper Null Termination: Heap Over Write) */
        strcat(stonesoup_path_buf,stonesoup_new_input);
        stonesoup_buf_ptr_len = strlen( *stonesoup_buf_ptr);
        for (; stonesoup_oc_i < stonesoup_buf_ptr_len; ++stonesoup_oc_i) {
   stonesoup_path_buf[stonesoup_oc_i] =
stonesoup_toupper(stonesoup_path_buf[stonesoup_oc_i]);
        }
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, variable_signed_integral,
"(stonesoup_path_buf[63])", (stonesoup_path_buf[63]), &(stonesoup_path_buf[63]),
"TRIGGER-STATE");
#endif /* STONESOUP_TRACE */
        stonesoup_printf("%s\n",stonesoup_path_buf);
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
    }
    if (stonesoup_path_buf != 0) {
        free(stonesoup_path_buf);
    }
    if (stonesoup_buf_ptr != 0) {
        free(stonesoup_buf_ptr);
    }
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_DEBUG
/******************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 ******************************************************************************/
int main(int argc,char *argv[])
{
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    char* tainted_buff = argv[1];

    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
    return 0;
```

```
}
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-190A Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1
#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1
#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1
#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1
#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1
#endif /* NOTHING DEFINED */

#if STONESOUP_SNIPPET_INCLUDES
/******************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 ******************************************************************************/
#include <stdlib.h> // malloc
#include <stdio.h> // vfprintf
#include <limits.h> // INT_MIN
/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_INCLUDES */
#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
/******************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 ******************************************************************************/
#include "output-handlers/handler.h"
#endif /* STONESOUP_SNIPPET_DEBUG */
```

```
int stonesoup_190_global_var = 0;
#endif /* STONESOUP_SNIPPET_SUPPORT */

#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 *****************************************************************************/
void weakness(char * stonesoup_tainted_buff) {
#define STONESOUP_TAINT_SOURCE stonesoup_tainted_buff
#endif /* STONESOUP_SNIPPET_DEBUG */
#if STONESOUP_SNIPPET_BODY_DECLARATIONS
    int stonesoup_tainted_int = 0;
    int stonesoup_output_counter = 0;
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_start, "CWE190", "A", "Integer Overflow or
Wraparound");
#endif /* STONESOUP_TRACE */
    stonesoup_tainted_int = atoi(STONESOUP_TAINT_SOURCE);
    if (stonesoup_tainted_int > 0) {
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
/* STONESOUP: CROSSOVER-POINT (Integer Overflow) */
        stonesoup_tainted_int = stonesoup_tainted_int * stonesoup_tainted_int;
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
        tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
        tracepoint(stonesoup_trace, variable_signed_integral, "stonesoup_tainted_int",
stonesoup_tainted_int, &stonesoup_tainted_int, "TRIGGER-STATE");
#endif /* STONESOUP_TRACE */
        while (stonesoup_tainted_int != 0) {
/* STONESOUP: TRIGGER-POINT (Integer Overflow) */
            if (stonesoup_tainted_int != INT_MIN) {
                stonesoup_tainted_int--;
            }
            /* Output only once every million iterations */
            if (stonesoup_output_counter == 0) {
                stonesoup_printf("evaluating input\n");
            }
            stonesoup_output_counter++;
            if (stonesoup_output_counter == 1000000) {
                stonesoup_output_counter = 0;
            }

            ++stonesoup_190_global_var;
            if (stonesoup_190_global_var >= INT_MAX) {
                stonesoup_190_global_var = 0;
            }
        }
#if STONESOUP_TRACE
```

```
        tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
    }
    stonesoup_printf("finished evaluating\n");
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 *****************************************************************************/
int main(int argc,char *argv[])
{
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    char* tainted_buff = argv[1];

    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
    exit(0);
}
```

```
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-191A Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1
#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1
#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1
#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1
#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1
#endif /* NOTHING DEFINED */

#if STONESOUP_SNIPPET_INCLUDES
/*******************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 ******************************************************************************/
#include <stdlib.h>
#include <stdio.h>
/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_INCLUDES */
#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 ******************************************************************************/
#include "output-handlers/handler.h"
#endif /* STONESOUP_SNIPPET_DEBUG */
int stonesoup_191_global_var = 0;
```

```
#endif /* STONESOUP_SNIPPET_SUPPORT */
#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 *****************************************************************************/
void weakness(char * stonesoup_tainted_buff) {
#define STONESOUP_TAINT_SOURCE stonesoup_tainted_buff
#endif /* STONESOUP_SNIPPET_DEBUG */
#if STONESOUP_SNIPPET_BODY_DECLARATIONS
 FILE * stonesoup_random_data = 0;
 char * stonesoup_buff = 0;
 unsigned long long stonesoup_num = 0;
 unsigned long long stonesoup_i = 0;
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_start, "CWE191", "A", "Integer Underflow
(Wrap or Wraparound)");
#endif /* STONESOUP_TRACE */
 stonesoup_random_data = fopen("/dev/urandom", "r");
 if(stonesoup_random_data != NULL){
  stonesoup_num = strtoull(STONESOUP_TAINT_SOURCE, NULL, 10);
  if(stonesoup_num > (unsigned long long) 0 ){
   if(stonesoup_num > (unsigned long long) 100 ){
    stonesoup_num = (unsigned long long) 100;
   }
#if STONESOUP_TRACE
            tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
            tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
            tracepoint(stonesoup_trace, variable_unsigned_integral, "stonesoup_num",
stonesoup_num, &stonesoup_num, "TRIGGER-STATE");
#endif /* STONESOUP_TRACE */
   /* STONESOUP: CROSSOVER-POINT (Integer Underflow) */
   /* STONESOUP: TRIGGER-POINT (Integer Underflow) */
   stonesoup_buff = malloc((stonesoup_num - (unsigned long long) 10) * sizeof(char
*));
   for(stonesoup_i = 0; stonesoup_i < stonesoup_num - (unsigned long long) 10;
stonesoup_i++){
    stonesoup_buff[stonesoup_i] = fgetc(stonesoup_random_data);
   }
#if STONESOUP_TRACE
            tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: AFTER");
            tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
  }
  stonesoup_printf("data is:  %s", stonesoup_buff);
  fclose(stonesoup_random_data);
  if(stonesoup_buff != NULL){
   free(stonesoup_buff);
  }
 }
#if STONESOUP_TRACE
```

```
    tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 ******************************************************************************/
int main(int argc,char *argv[])
{
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    char* tainted_buff = argv[1];

    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
```

```
    exit(0);
}
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-191B Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1
#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1
#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1
#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1
#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1
#endif /* NOTHING DEFINED */

#if STONESOUP_SNIPPET_INCLUDES
/*****************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 *****************************************************************************/
#include <stdlib.h>
#include <stdio.h>
/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_INCLUDES */
#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 *****************************************************************************/
#include "output-handlers/handler.h"
#endif /* STONESOUP_SNIPPET_DEBUG */
```

```
int stonesoup_191_global_var = 0;
#endif /* STONESOUP_SNIPPET_SUPPORT */
#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 *****************************************************************************/
void weakness(char * stonesoup_tainted_buff) {
#define STONESOUP_TAINT_SOURCE stonesoup_tainted_buff
#endif /* STONESOUP_SNIPPET_DEBUG */
#if STONESOUP_SNIPPET_BODY_DECLARATIONS
        char stonesoup_buff[128];
        short stonesoup_max_size = 128;
        short stonesoup_remaining_space = 0;
        int stonesoup_size = 0;
        FILE * stonesoup_file = 0;
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, weakness_start, "CWE191", "B", "Integer Underflow
(Wrap or Wraparound)");
#endif /* STONESOUP_TRACE */
        stonesoup_file = fopen(STONESOUP_TAINT_SOURCE, "r");
        if(stonesoup_file != NULL){
                fseek(stonesoup_file, 0, SEEK_END);
                stonesoup_size = ftell(stonesoup_file);
                fseek(stonesoup_file, 0, SEEK_SET);
#if STONESOUP_TRACE
                tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
                /* STONESOUP: CROSSOVER-POINT (Integer Underflow) */
                stonesoup_remaining_space = stonesoup_max_size - stonesoup_size;
#if STONESOUP_TRACE
                tracepoint(stonesoup_trace, variable_unsigned_integral,
"stonesoup_remaining_space", stonesoup_remaining_space, &stonesoup_remaining_space,
"CROSSOVER-STATE");
                tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
                tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
                /* STONESOUP: TRIGGER-POINT (Integer Underflow) */
                if(stonesoup_remaining_space > 0){
                        fscanf(stonesoup_file, "%s", stonesoup_buff);
                        stonesoup_printf("Contents of file: %s\n", stonesoup_buff);
                }
#if STONESOUP_TRACE
                tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
                fclose(stonesoup_file);
        }
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
```

```
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 *****************************************************************************/
int main(int argc,char *argv[])
{
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    char* tainted_buff = argv[1];

    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
```

Approved for public release; distribution unlimited.

```
    exit(0);
}
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-194A Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1
#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1
#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1
#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1
#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1
#endif /* NOTHING DEFINED */

#if STONESOUP_SNIPPET_INCLUDES
/******************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 ******************************************************************************/
#include <stdlib.h> /* malloc */
#include <string.h> /* memset */
#include <stdio.h> /* FILE */
#include <limits.h> /* INT_MIN */
/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_INCLUDES */
#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
/******************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 ******************************************************************************/
```

```
#include "output-handlers/handler.h"
#endif /* STONESOUP_SNIPPET_DEBUG */
short stonesoup_get_int_value(char *ss_tainted_buff)
{
#if STONESOUP_TRACE
  tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE,
"stonesoup_get_int_value");
#endif /* STONESOUP_TRACE */
  short to_short = 0;
  int tainted_int = 0;
  tainted_int = atoi(ss_tainted_buff);
  if (tainted_int != 0) {
    if (tainted_int > 30000)
      tainted_int = 30000;
    if (tainted_int < -30000)
      tainted_int = -30000;
    to_short = ((short )tainted_int);
  }
  return to_short;
}
#endif /* STONESOUP_SNIPPET_SUPPORT */

#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 *****************************************************************************/
void weakness(char * stonesoup_tainted_buff) {
#define STONESOUP_TAINT_SOURCE stonesoup_tainted_buff
#endif /* STONESOUP_SNIPPET_DEBUG */
#define STONESOUP_EXPRESSION_1 "/opt/stonesoup/workspace/testData/myfile.txt"
#if STONESOUP_SNIPPET_BODY_DECLARATIONS
    unsigned int stonesoup_to_unsign = 0;
    char *stonesoup_buff = 0;
    FILE *stonesoup_file = 0;
    int stonesoup_counter = 0;
    int stonesoup_bytes_read = 0;
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_start, "CWE194", "A", "Unexpected Sign
Extension");
#endif /* STONESOUP_TRACE */
    stonesoup_buff = ((char *)(malloc(30000  * sizeof(char ))));
    if (stonesoup_buff == 0) {
        stonesoup_printf("Error: Failed to allocate memory\n");
        exit(1);
    }
    memset(stonesoup_buff, 0, 30000);
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
/* STONESOUP: CROSSOVER-POINT (Unexpected Sign Extension) */
```

```
    stonesoup_to_unsign = stonesoup_get_int_value(STONESOUP_TAINT_SOURCE);
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, variable_buffer, "STONESOUP_TAINT_SOURCE",
STONESOUP_TAINT_SOURCE, "CROSSOVER-STATE");
    tracepoint(stonesoup_trace, variable_unsigned_integral, "stonesoup_to_unsign",
stonesoup_to_unsign, &stonesoup_to_unsign, "CROSSOVER-STATE");
    tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
    stonesoup_file = fopen(STONESOUP_EXPRESSION_1,"r");
    if (stonesoup_file != 0) {
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
        while (((unsigned int )stonesoup_counter) < stonesoup_to_unsign) {
            /* STONESOUP: TRIGGER-POINT (Unexpected Sign Extension) */
            stonesoup_bytes_read = fread(&stonesoup_buff[stonesoup_counter],
                sizeof(char), 1000, stonesoup_file);
            if (stonesoup_bytes_read == 0) {
                break;
            }
            stonesoup_counter += stonesoup_bytes_read;
        }
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
        fclose(stonesoup_file);
        stonesoup_buff[stonesoup_to_unsign] = '\0';
        stonesoup_printf("buff is %d long, and has contents: %s
\n",strlen(stonesoup_buff), stonesoup_buff);
    } else {
        stonesoup_printf("Cannot open file %s\n", STONESOUP_EXPRESSION_1);
    }
    if (stonesoup_buff != 0) {
        free(stonesoup_buff);
    }
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 *******************************************************************************/
int main(int argc,char *argv[])
{
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    char* tainted_buff = argv[1];

    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
```

```
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
}
```

```
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-195A Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1
#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1
#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1
#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1
#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1
#endif /* NOTHING DEFINED */

#if STONESOUP_SNIPPET_INCLUDES
/******************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 ******************************************************************************/
#include <stdlib.h> // exit
#include <stdio.h> // vfprintf
#include <string.h> // memset
/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_INCLUDES */
#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
/******************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 ******************************************************************************/
#include "output-handlers/handler.h"
#endif /* STONESOUP_SNIPPET_DEBUG */
```

```
int stonesoup_returnChunkSize(void *dest,void *src)
{
#if STONESOUP_TRACE
  tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE,
"stonesoup_returnChunkSize");
#endif /* STONESOUP_TRACE */
  if (strlen(dest) < strlen(src)) {
/* STONESOUP: CROSSOVER-POINT (Signed To Unsigned Conversion Error) */
    return -1;
  }
  return strlen(dest);
}
#endif /* STONESOUP_SNIPPET_SUPPORT */
#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 ******************************************************************************/
void weakness(char * stonesoup_tainted_buff) {
#define STONESOUP_TAINT_SOURCE stonesoup_tainted_buff
#endif /* STONESOUP_SNIPPET_DEBUG */
#if STONESOUP_SNIPPET_BODY_DECLARATIONS
    const int stonesoup_MAXLEN = 16;
    char stonesoup_dest[stonesoup_MAXLEN];
    size_t stonesoup_size_var = 0;
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_start, "CWE195", "A", "Signed to Unsigned
Conversion Error");
#endif /* STONESOUP_TRACE */
    memset(stonesoup_dest,'x',stonesoup_MAXLEN);
    stonesoup_dest[stonesoup_MAXLEN - 1] = '\0';
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
    stonesoup_size_var = stonesoup_returnChunkSize(stonesoup_dest,
STONESOUP_TAINT_SOURCE);
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
    tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
    tracepoint(stonesoup_trace, variable_signed_integral, "stonesoup_size_var",
stonesoup_size_var, &stonesoup_size_var, "TRIGGER-STATE");
#endif /* STONESOUP_TRACE */
/* STONESOUP: TRIGGER-POINT (Signed To Unsigned Conversion Error) */
    if (stonesoup_size_var > 0)
        memcpy(stonesoup_dest, STONESOUP_TAINT_SOURCE, stonesoup_size_var);
    stonesoup_printf("%s\n",stonesoup_dest);
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: AFTER");
    tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
```

```
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 *****************************************************************************/
int main(int argc,char *argv[])
{
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    char* tainted_buff = argv[1];

    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
    exit(0);
}
```

```
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-196A Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1
#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1
#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1
#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1
#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1
#endif /* NOTHING DEFINED */

#if STONESOUP_SNIPPET_INCLUDES
/*****************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 *****************************************************************************/
#include <stdlib.h> // malloc
#include <stdio.h> // vfprintf
#include <string.h> // memset
/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_INCLUDES */
#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 *****************************************************************************/
#include "output-handlers/handler.h"
#endif /* STONESOUP_SNIPPET_DEBUG */
```

```
unsigned int stonesoup_get_size(char *ss_tainted)
{
#if STONESOUP_TRACE
  tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE,
"stonesoup_get_size");
#endif /* STONESOUP_TRACE */
  unsigned long uns_int = 0UL;
  uns_int = strtoul(ss_tainted,0,0);
  if (uns_int > ((unsigned long )4294967295U) ||
      uns_int == 0)
    uns_int = 1U;
  return (unsigned int )uns_int;
}
#endif /* STONESOUP_SNIPPET_SUPPORT */
#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 ******************************************************************************/
void weakness(char * stonesoup_tainted_buff) {
#define STONESOUP_TAINT_SOURCE stonesoup_tainted_buff
#endif /* STONESOUP_SNIPPET_DEBUG */
#define STONESOUP_EXPRESSION_1 "SS_BUFF"
#if STONESOUP_SNIPPET_BODY_DECLARATIONS
    char *stonesoup_other_buff = 0;
    int stonesoup_size = 0;
    int stonesoup_num = 0;
    char stonesoup_buff[200] = {0};
    int stonesoup_output_counter = 0;
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_start, "CWE196", "A", "Unsigned to Signed
Conversion Error");
#endif /* STONESOUP_TRACE */
    if (strlen(STONESOUP_TAINT_SOURCE) > 0 &&
        STONESOUP_TAINT_SOURCE[0] == '-') {
        stonesoup_printf("Negative number given as input\n");
    } else {
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
        /* STONESOUP: CROSSOVER-POINT (Unsigned To Signed Conversion Error) */
        stonesoup_num = stonesoup_get_size(STONESOUP_TAINT_SOURCE);
        stonesoup_other_buff = getenv(STONESOUP_EXPRESSION_1);
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, variable_signed_integral, "stonesoup_num",
stonesoup_num, &stonesoup_num, "CROSSOVER-STATE");
        tracepoint(stonesoup_trace, variable_buffer, "stonesoup_other_buff",
stonesoup_other_buff, "CROSSOVER-STATE");
        tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
        if (stonesoup_other_buff != 0) {
```

```
            strncpy(stonesoup_buff, stonesoup_other_buff, (sizeof(stonesoup_buff) -
1)/(sizeof(char)));
            stonesoup_size = ((int )(strlen(stonesoup_buff)));
#if STONESOUP_TRACE
            tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
            /* STONESOUP: TRIGGER-POINT (Unsigned To Signed Conversion Error) */
            while (stonesoup_num < stonesoup_size) {
                /* Output only once every million iterations */
                if (stonesoup_output_counter == 0) {
                    stonesoup_printf("evaluating input\n");
                }
                stonesoup_output_counter++;
                if (stonesoup_output_counter == 1000000) {
                    stonesoup_output_counter = 0;
                }

                if (stonesoup_num > 0)
                    ++stonesoup_num;
            }
#if STONESOUP_TRACE
            tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
        } else {
            stonesoup_printf("Missing value for other_buff\n");
        }
        stonesoup_printf("finished evaluating\n");
    }
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_DEBUG
/******************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 ******************************************************************************/
int main(int argc,char *argv[])
{
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    char* tainted_buff = argv[1];

    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
    exit(0);
}
```

```
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-197A Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1
#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1
#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1
#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1
#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1
#endif /* NOTHING DEFINED */

#if STONESOUP_SNIPPET_INCLUDES
/*****************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 *****************************************************************************/
#include <stdlib.h> // exit
#include <stdio.h> // vfprintf
#include <string.h> // memset
/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_INCLUDES */
#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 *****************************************************************************/
#include "output-handlers/handler.h"
#endif /* STONESOUP_SNIPPET_DEBUG */
```

```
struct stonesoup_struct_data {
  char *buffer_member;
  unsigned short size_member;
};
struct stonesoup_struct_data *stonesoup_init_data(long number_param)
{
#if STONESOUP_TRACE
  tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE,
"stonesoup_init_data");
#endif /* STONESOUP_TRACE */
  struct stonesoup_struct_data *init_data_ptr = 0;
  init_data_ptr = ((struct stonesoup_struct_data *)(malloc(sizeof(struct
stonesoup_struct_data ))));
  if (init_data_ptr == 0)
    return 0;
  init_data_ptr -> size_member = 0;
#if STONESOUP_TRACE
  tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
/* STONESOUP: CROSSOVER-POINT (Numerical Truncation Error) */
  init_data_ptr -> size_member = number_param;
  init_data_ptr -> buffer_member = ((char *)(malloc(sizeof(char ) * init_data_ptr ->
size_member)));
#if STONESOUP_TRACE
  tracepoint(stonesoup_trace, variable_signed_integral, "number_param", number_param,
&number_param, "CROSSOVER-STATE");
  tracepoint(stonesoup_trace, variable_signed_integral,
"(init_data_ptr->size_member)", (init_data_ptr->size_member),
&(init_data_ptr->size_member), "CROSSOVER-STATE");
  tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
  if (init_data_ptr -> buffer_member == 0) {
    free(init_data_ptr);
    return 0;
  }
  memset(init_data_ptr -> buffer_member,'a',init_data_ptr -> size_member);
  init_data_ptr -> buffer_member[init_data_ptr -> size_member - 1] = 0;
  return init_data_ptr;
}
#endif /* STONESOUP_SNIPPET_SUPPORT */
#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 *****************************************************************************/
void weakness(char * stonesoup_tainted_buff) {
#define STONESOUP_TAINT_SOURCE stonesoup_tainted_buff
#endif /* STONESOUP_SNIPPET_DEBUG */
#if STONESOUP_SNIPPET_BODY_DECLARATIONS
    long stonesoup_number;
    struct stonesoup_struct_data *stonesoup_data = 0;
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_BODY_STATEMENTS
```

```
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_start, "CWE197", "A", "Numeric Truncation
Error");
#endif /* STONESOUP_TRACE */
    stonesoup_number = strtol(STONESOUP_TAINT_SOURCE,0U,10);
    if (stonesoup_number > 0) {
        stonesoup_data = stonesoup_init_data(stonesoup_number);
        if (stonesoup_data != 0) {
#if STONESOUP_TRACE
          tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
/* STONESOUP: TRIGGER-POINT (Numerical Truncation Error) */
          memset(stonesoup_data -> buffer_member, 98, stonesoup_number);
#if STONESOUP_TRACE
          tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
          stonesoup_data -> buffer_member[stonesoup_number - 1] = 0;
          stonesoup_printf("%s\n", stonesoup_data -> buffer_member);
          if (stonesoup_data -> buffer_member != 0U)
            free(stonesoup_data -> buffer_member);
          if (stonesoup_data != 0U)
            free(stonesoup_data);
        }
    } else {
        stonesoup_printf("Input is less than or equal to 0\n");
    }
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_DEBUG
/******************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 ******************************************************************************/
int main(int argc,char *argv[])
{
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    char* tainted_buff = argv[1];

    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
    exit(0);
}
```

```
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-363A Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1

#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1

#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1

#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1

#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1

#endif /* NOTHING DEFINED */


#if STONESOUP_SNIPPET_INCLUDES
/******************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 ******************************************************************************/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>

/*
 *  * Include tracepoint events.
 *   */
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */

#endif /* STONESOUP_SNIPPET_INCLUDES */
```

```
#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 *****************************************************************************/
#include "output-handlers/handler.h"

#endif /* STONESOUP_SNIPPET_DEBUG */

int stonesoup_comp (const void * a, const void * b)
{
    if (a > b) {
        return -1;
    }
    else if (a < b) {
        return 1;
    }
    else {
        return 0;
    }
}

int stonesoup_pmoc (const void * a, const void * b)
{
    return -1 * stonesoup_comp(a, b);
}

void stonesoup_readFile(char *filename) {
    FILE *fifo;
    char ch;
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE,
"stonesoup_readFile");
#endif

    fifo = fopen(filename, "r");

    if (fifo != NULL) {
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, trace_point, "Reading from FIFO");
#endif
        while ((ch = fgetc(fifo)) != EOF) {
            stonesoup_printf("%c", ch);
        }

        fclose(fifo);
    }
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "Exiting readFile");
#endif
}

void waitForChange(char* file, char* sleepFile) {
```

```
    int fd;
    char filename[500] = {0};
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE,
"stonesoup_waitForChange");
#endif

    stonesoup_printf("In waitForChange\n");

    strcat(filename, file);
    strcat(filename, ".pid");

    if ((fd = open(filename, O_CREAT|O_WRONLY, 0666)) == -1) {
        stonesoup_printf("Error opening file.");
    }
    else {
        if (write(fd, "q", sizeof(char)) == -1) {
            stonesoup_printf("Error writing to file.");
        }
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, trace_point, "Wrote .pid file");
#endif

        if (close(fd) == -1) {
#if STONESOUP_TRACE
            tracepoint(stonesoup_trace, trace_error, "Error closing file.");
#endif
            stonesoup_printf("Error closing file.");
        }
        stonesoup_readFile(sleepFile);
    }
}

int stonesoup_path_is_relative(char *path) {
    char *chr = 0;
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE,
"stonesoup_path_is_relative");
#endif
    chr = strchr(path,'/');
    if (chr == 0) {
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, trace_point, "Path is relative");
#endif
        stonesoup_printf("Path is relative\n");
        return 1;
    } else {
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, trace_point, "Path is not relative");
#endif
        stonesoup_printf("Path is not relative\n");
        return 0;
    }
}

char * stonesoup_get_absolute_path(char * path) {
    char * abs_path = malloc (sizeof(char) * (strlen(STONESOUP_TESTDATA) *
strlen(path) + 1));
#if STONESOUP_TRACE
```

```
    tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE,
"stonesoup_get_absolute_path");
#endif
    if (abs_path == NULL) {
        stonesoup_printf("Cannot allocate memory for path\n");
    } else {
        stonesoup_printf("Creating absolute path\n");
        strcpy(abs_path, STONESOUP_TESTDATA);
        strcat(abs_path, path);
    }
    return abs_path;
}

int stonesoup_isSymLink(char *file) {
    struct stat statbuf;
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE,
"stonesoup_isSymLink");
#endif

    if (lstat(file, &statbuf) < 0) {    /* if error occured */
        stonesoup_printf("Error accessing path.\n");
        return 1;                       /* just end program */
    }

    if (S_ISLNK(statbuf.st_mode) == 1) {
        stonesoup_printf("Path is symlink.\n");
        return 1;
    }
    stonesoup_printf("Path is valid.\n");
    return 0;
}

int stonesoup_path_is_not_symlink(char * abs_path) {
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE,
"stonesoup_path_is_not_symlink");
#endif
    return (stonesoup_isSymLink(abs_path) == 0);
}

#endif /* STONESOUP_SNIPPET_SUPPORT */

#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 ******************************************************************************/
void weakness(char * stonesoup_tainted_buff) {
#define STONESOUP_TAINT_SOURCE stonesoup_tainted_buff
#endif /* STONESOUP_SNIPPET_DEBUG */

#if STONESOUP_SNIPPET_BODY_DECLARATIONS
```

```
    int stonesoup_size = 0;
    FILE *stonesoup_file = 0;
    char *stonesoup_buffer = 0;
    char *stonesoup_str = 0;
    char *stonesoup_abs_path = 0;
    char *stonesoup_sleep_file = 0;
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_start, "CWE363", "A", "Race Condition
Enabling Link Following");
#endif /* STONESOUP_TRACE */
    stonesoup_str = malloc(sizeof(char) * (strlen(STONESOUP_TAINT_SOURCE) + 1));
    stonesoup_sleep_file = malloc(sizeof(char) * (strlen(STONESOUP_TAINT_SOURCE) +
1));
    if (stonesoup_str != NULL && stonesoup_sleep_file != NULL &&
        (sscanf(STONESOUP_TAINT_SOURCE, "%s %s",
                stonesoup_sleep_file,
                stonesoup_str) == 2) &&
        (strlen(stonesoup_str) != 0) &&
        (strlen(stonesoup_sleep_file) != 0))
    {
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, variable_buffer, "stonesoup_sleep_file",
stonesoup_sleep_file, "INITIAL-STATE");
        tracepoint(stonesoup_trace, variable_buffer, "stonesoup_str", stonesoup_str,
"INITIAL-STATE");
#endif
        if (stonesoup_path_is_relative(stonesoup_str)) {
            stonesoup_abs_path = stonesoup_get_absolute_path(stonesoup_str);
            if (stonesoup_abs_path != NULL) {
                if (stonesoup_path_is_not_symlink(stonesoup_abs_path)) {
#if STONESOUP_TRACE
                    tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT:
BEFORE");
#endif
                    /* STONESOUP: CROSSOVER-POINT (race condition enabling link
following) */
                    waitForChange(stonesoup_abs_path, stonesoup_sleep_file);
                    stonesoup_file = fopen(stonesoup_abs_path,"rb");
#if STONESOUP_TRACE
                    tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT:
AFTER");
#endif
                    if (stonesoup_file != 0) {
                        fseek(stonesoup_file,0,2);
                        stonesoup_size = ftell(stonesoup_file);
                        rewind(stonesoup_file);
                        stonesoup_buffer = ((char *)(malloc(sizeof(char ) *
(stonesoup_size + 1))));
                        if (stonesoup_buffer) {
#if STONESOUP_TRACE
                            tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT:
BEFORE");
#endif
                            /* STONESOUP: TRIGGER-POINT (race condition enabling link
following) */
                            fread(stonesoup_buffer,sizeof(char
),stonesoup_size,stonesoup_file);
```

```
                            stonesoup_buffer[stonesoup_size] = '\0';
                            stonesoup_printf(stonesoup_buffer);
                            fclose(stonesoup_file);
                            free(stonesoup_buffer);
#if STONESOUP_TRACE
                            tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT:
AFTER");
#endif
                        }
                    }
                }
                free (stonesoup_abs_path);
            }
        }
        free(stonesoup_str);
    } else {
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, trace_point, "Error parsing input.");
#endif
        stonesoup_printf("Error parsing input.\n");
    }
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */

#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 *****************************************************************************/
int main(int argc,char *argv[])
{
    char* tainted_buff;
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    tainted_buff = argv[1];

    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
```

```
    return 0;
}
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-367A Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1

#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1

#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1

#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1

#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1

#endif /* NOTHING DEFINED */


#if STONESOUP_SNIPPET_INCLUDES
/*****************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 *****************************************************************************/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>

/*
 * Include tracepoint events.
 */
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */

#endif /* STONESOUP_SNIPPET_INCLUDES */
```

```
#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 *****************************************************************************/
#include "output-handlers/handler.h"

#endif /* STONESOUP_SNIPPET_DEBUG */

int stonesoup_comp (const void * a, const void * b)
{
    if (a > b) {
        return -1;
    }
    else if (a < b) {
        return 1;
    }
    else {
        return 0;
    }
}

int stonesoup_pmoc (const void * a, const void * b)
{
    return -1 * stonesoup_comp(a, b);
}

void stonesoup_readFile(char *filename) {
    FILE *fifo;
    char ch;
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE,
"stonesoup_readFile");
#endif

    fifo = fopen(filename, "r");

    if (fifo != NULL) {
        while ((ch = fgetc(fifo)) != EOF) {
            stonesoup_printf("%c", ch);
        }

        fclose(fifo);
    }
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "Finished reading from sync file.");
#endif
}

void waitForChange(char* file, char* sleepFile) {
    int fd;
    char filename[500] = {0};
#if STONESOUP_TRACE
```

Approved for public release; distribution unlimited.

```
    tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE,
"stonesoup_waitForChange");
#endif

    stonesoup_printf("In waitForChange\n");

    strcat(filename, file);
    strcat(filename, ".pid");

    if ((fd = open(filename, O_CREAT|O_WRONLY, 0666)) == -1) {
        stonesoup_printf("Error opening file.");
    }
    else {
        if (write(fd, "q", sizeof(char)) == -1) {
#if STONESOUP_TRACE
            tracepoint(stonesoup_trace, trace_error, "Error writing to file.");
#endif
            stonesoup_printf("Error writing to file.");
        }
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, trace_point, "Wrote .pid file.");
#endif
        if (close(fd) == -1) {
#if STONESOUP_TRACE
            tracepoint(stonesoup_trace, trace_error, "Error closing file.");
#endif
            stonesoup_printf("Error closing file.");
        }
        stonesoup_readFile(sleepFile);
    }
}

int stonesoup_is_valid(char *path)
{
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE,
"stonesoup_is_valid");
#endif
    if(access(path, F_OK) != -1) {
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, trace_point, "Path is accessible");
#endif
        stonesoup_printf("Path is accessible\n");
        return 1;
    }
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "Path is not accessible");
#endif
    stonesoup_printf("Path is not accessible\n");

    return 0;
}

int stonesoup_path_is_relative(char *path) {
    char *chr = 0;
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE,
"stonesoup_path_is_relative");
#endif
```

```
    chr = strchr(path,'/');
    if (chr == 0) {
        stonesoup_printf("Path is relative\n");
        return 1;
    } else {
        stonesoup_printf("Path is not relative\n");
        return 0;
    }
}

char * stonesoup_get_absolute_path(char * path) {
    char * abs_path = malloc (sizeof(char) * (strlen(STONESOUP_TESTDATA) *
strlen(path) + 1));
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE,
"stonesoup_get_absolute_path");
#endif
    if (abs_path == NULL) {
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, trace_error, "Cannot allocate memory for path");
#endif
        stonesoup_printf("Cannot allocate memory for path\n");
    } else {
        stonesoup_printf("Creating absolute path\n");
        strcpy(abs_path, STONESOUP_TESTDATA);
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, variable_buffer, "abs_path", abs_path, "Generated
absolute path");
#endif
        strcat(abs_path, path);
    }
    return abs_path;
}

#endif /* STONESOUP_SNIPPET_SUPPORT */

#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 ******************************************************************************/
void weakness(char * stonesoup_tainted_buff) {
#define STONESOUP_TAINT_SOURCE stonesoup_tainted_buff
#endif /* STONESOUP_SNIPPET_DEBUG */
#if STONESOUP_SNIPPET_BODY_DECLARATIONS
    int stonesoup_size = 0;
    FILE *stonesoup_file = 0;
    char *stonesoup_buffer = 0;
    char *stonesoup_str = 0;
    char *stonesoup_abs_path = 0;
    char *stonesoup_sleep_file = 0;
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_BODY_STATEMENTS
```

```
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_start, "CWE367", "A", "Time of Check Time of
Use Race Condition");
#endif /* STONESOUP_TRACE */
    stonesoup_str = malloc(sizeof(char) * (strlen(STONESOUP_TAINT_SOURCE) + 1));
    stonesoup_sleep_file = malloc(sizeof(char) * (strlen(STONESOUP_TAINT_SOURCE) +
1));
    if (stonesoup_str != NULL && stonesoup_sleep_file != NULL &&
        (sscanf(STONESOUP_TAINT_SOURCE, "%s %s",
                stonesoup_sleep_file,
                stonesoup_str) == 2) &&
        (strlen(stonesoup_str) != 0) &&
        (strlen(stonesoup_sleep_file) != 0))
    {
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, variable_buffer, "stonesoup_sleep_file",
stonesoup_sleep_file, "INITIAL-STATE");
        tracepoint(stonesoup_trace, variable_buffer, "stonesoup_str", stonesoup_str,
"INITIAL-STATE");
#endif
        if (stonesoup_path_is_relative(stonesoup_str)) {
            stonesoup_abs_path = stonesoup_get_absolute_path(stonesoup_str);
            if (stonesoup_abs_path != NULL) {
               if (stonesoup_is_valid(stonesoup_abs_path)) {
#if STONESOUP_TRACE
                  tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
#endif
                    /* STONESOUP: CROSSOVER-POINT (Time of Check, Time of Use) */
                    waitForChange(stonesoup_abs_path, stonesoup_sleep_file);
#if STONESOUP_TRACE
                   tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
                   tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
#endif
                    /* STONESOUP: TRIGGER-POINT (Time of Check, Time of Use) */
                    stonesoup_file = fopen(stonesoup_abs_path,"rb");
                    fseek(stonesoup_file,0,2);
                    stonesoup_size = ftell(stonesoup_file);
                    rewind(stonesoup_file);
                    stonesoup_buffer = ((char *)(malloc(sizeof(char ) *
(stonesoup_size + 1))));
                    if (stonesoup_buffer != NULL) {
                        fread(stonesoup_buffer,sizeof(char
),stonesoup_size,stonesoup_file);
                        stonesoup_buffer[stonesoup_size] = '\0';
                        stonesoup_printf(stonesoup_buffer);
                        free(stonesoup_buffer);
                    }
#if STONESOUP_TRACE
                   tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: AFTER");
#endif
                    fclose(stonesoup_file);
                }
            }
            free(stonesoup_abs_path);
        }
        free(stonesoup_str);
    } else {
#if STONESOUP_TRACE
       tracepoint(stonesoup_trace, trace_error, "Error parsing input");
```

```
#endif
        stonesoup_printf("Error parsing input.\n");
    }
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */

#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 *******************************************************************************/
int main(int argc,char *argv[])
{
    char* tainted_buff;
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    tainted_buff = argv[1];

    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
```

```
    exit(0);
}
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-369A Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1
#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1
#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1
#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1
#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1
#endif /* NOTHING DEFINED */

#if STONESOUP_SNIPPET_INCLUDES
/*****************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 *****************************************************************************/
#include <stdlib.h> // exit
#include <stdio.h> // vfprintf
#include <string.h> // memset
/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_INCLUDES */
#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 *****************************************************************************/
#include "output-handlers/handler.h"
```

```
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_SUPPORT */
#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 *****************************************************************************/
void weakness(char * stonesoup_tainted_buff) {
#define STONESOUP_TAINT_SOURCE stonesoup_tainted_buff
#endif /* STONESOUP_SNIPPET_DEBUG */
#if STONESOUP_SNIPPET_BODY_DECLARATIONS
    float stonesoup_quotient;
    int stonesoup_mod = 0;
    int stonesoup_input;
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_start, "CWE369", "A", "Divide By Zero");
#endif /* STONESOUP_TRACE */
    stonesoup_input = atoi(STONESOUP_TAINT_SOURCE);
    if (stonesoup_input != 0) {
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
        /* STONESOUP: CROSSOVER-POINT (Divide By Zero) */
        stonesoup_mod = stonesoup_input % 4;
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, variable_signed_integral, "stonesoup_input",
stonesoup_input, &stonesoup_input, "CROSSOVER-STATE");
        tracepoint(stonesoup_trace, variable_signed_integral, "stonesoup_mod",
stonesoup_mod, &stonesoup_mod, "CROSSOVER-STATE");
        tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
        tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
        /* STONESOUP: TRIGGER-POINT (Divide By Zero) */
        stonesoup_quotient = 1024 / stonesoup_mod;
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
        stonesoup_printf("%f\n", stonesoup_quotient);
    } else {
        stonesoup_printf("Input value is 0, or not a number\n");
    }
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
```

```
 * Main
 *
 * This only exists to support direct debugging.
 ******************************************************************************/
int main(int argc,char *argv[])
{
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    char* tainted_buff = argv[1];

    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
    exit(0);
```

Approved for public release; distribution unlimited.

```
}
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-412A Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1

#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1

#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1

#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1

#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1

#endif /* NOTHING DEFINED */


#if STONESOUP_SNIPPET_INCLUDES
/*****************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 *****************************************************************************/
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

/*
 * Include tracepoint events.
 */
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */

#endif /* STONESOUP_SNIPPET_INCLUDES */

#if STONESOUP_SNIPPET_SUPPORT
```

```
#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 ******************************************************************************/
#include "output-handlers/handler.h"

#endif /* STONESOUP_SNIPPET_DEBUG */

#endif /* STONESOUP_SNIPPET_SUPPORT */

#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 ******************************************************************************/
void weakness(char * taintedValue) {
#define STONESOUP_TAINT_SOURCE taintedValue
#endif /* STONESOUP_SNIPPET_DEBUG */
#if STONESOUP_SNIPPET_BODY_DECLARATIONS
    int stonesoup_fd;
    char *stonesoup_extension = ".lck";
    char *stonesoup_thelockfile;
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_start, "CWE412", "A", "Unrestricted
Externally Accessible Lock");
#endif /* STONESOUP_TRACE */
    stonesoup_thelockfile =  (char *)malloc(strlen(STONESOUP_TESTDATA) +
strlen(STONESOUP_TAINT_SOURCE) +
                                  strlen(stonesoup_extension) + 1);
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, variable_buffer, "stonesoup_thelockfile",
stonesoup_thelockfile, "INITIAL-STATE");
#endif
    if (stonesoup_thelockfile) {
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
#endif
        /* STONESOUP: CROSSOVER-POINT (Deadlock) */
  strcpy(stonesoup_thelockfile, STONESOUP_TESTDATA);
     strcat(stonesoup_thelockfile, STONESOUP_TAINT_SOURCE);
     strcat(stonesoup_thelockfile, stonesoup_extension);
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
#endif
```

```
      /* Check if the path is accessible */
      if (access(STONESOUP_TESTDATA,  R_OK && W_OK) == -1) {
       stonesoup_printf("Error accessing testData directory\n");
      } else {

            stonesoup_printf("Grabbing lock file: %s\n", stonesoup_thelockfile);
#if STONESOUP_TRACE
            tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
#endif
            /* Create new "lock" file. If it already exists, it means the lock is
taken and we
               need to wait for it to be released. */
       /* STONESOUP: TRIGGER-POINT (Deadlock) */
            while ((stonesoup_fd = open(stonesoup_thelockfile, O_CREAT | O_EXCL, 0644)
== -1) ) {
                sleep(0);
            }
#if STONESOUP_TRACE
            tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: AFTER");
#endif

            stonesoup_printf("Made it to the critical section\n");

            if (stonesoup_fd != -1) {
                close(stonesoup_fd);
                unlink(stonesoup_thelockfile);
            }
            free (stonesoup_thelockfile);
        }
    }
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */

#if STONESOUP_SNIPPET_DEBUG
/******************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 ******************************************************************************/
int main(int argc,char *argv[])
{
    char* tainted_buff;
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    tainted_buff = argv[1];

    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
```

```
    exit(0);
}
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-414A Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1

#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1

#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1

#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1

#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1

#endif /* NOTHING DEFINED */


#if STONESOUP_SNIPPET_INCLUDES
/*****************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 *****************************************************************************/
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <string.h>

/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */

#endif /* STONESOUP_SNIPPET_INCLUDES */

#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
```

```
/******************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 ******************************************************************************/
#include "output-handlers/handler.h"

#endif /* STONESOUP_SNIPPET_DEBUG */

struct stonesoup_data {
    int qsize;
    int data_size;
    char *data;
    char *file1;
    char *file2;
};

pthread_mutex_t stonesoup_mutex;

int stonesoup_comp (const void * a, const void * b)
{
    if (a > b) {
        return -1;
    }
    else if (a < b) {
        return 1;
    }
    else {
        return 0;
    }
}

int stonesoup_pmoc (const void * a, const void * b)
{
    return -1 * stonesoup_comp(a, b);
}

void stonesoup_readFile(char *filename) {
    FILE *fifo;
    char ch;

#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE,
"stonesoup_readFile");
#endif /* STONESOUP_TRACE */

    fifo = fopen(filename, "r");

    if (fifo != NULL) {
        while ((ch = fgetc(fifo)) != EOF) {
            stonesoup_printf("%c", ch);
        }

        fclose(fifo);
    }
```

```
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "Finished reading sync file.");
#endif /* STONESOUP_TRACE */
}

void *toCap (void *data) {
    struct stonesoup_data *stonesoupData = (struct stonesoup_data*)data;
    int stonesoup_i;
    int *stonesoup_arr;

#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE, "toCap");
#endif /* STONESOUP_TRACE */

    pthread_mutex_lock(&stonesoup_mutex);

    stonesoup_readFile(stonesoupData->file1);

    stonesoup_arr = malloc(sizeof(int) * stonesoupData->qsize);
    for (stonesoup_i = 0; stonesoup_i < stonesoupData->qsize; stonesoup_i++) {
        stonesoup_arr[stonesoup_i] = stonesoupData->qsize - stonesoup_i;
    }
    qsort(stonesoup_arr, stonesoupData->qsize, sizeof(int), &stonesoup_comp);
    free(stonesoup_arr);

#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
    tracepoint(stonesoup_trace, variable_signed_integral, "stonesoupData->qsize",
stonesoupData->qsize, &(stonesoupData->qsize), "TRIGGER-STATE");
    tracepoint(stonesoup_trace, variable_buffer, "stonesoupData->data",
stonesoupData->data, "TRIGGER-STATE");
    tracepoint(stonesoup_trace, variable_buffer, "stonesoupData->file1",
stonesoupData->file1, "TRIGGER-STATE");
    tracepoint(stonesoup_trace, variable_buffer, "stonesoupData->file2",
stonesoupData->file2, "TRIGGER-STATE");
#endif /* STONESOUP_TRACE */
    for(stonesoup_i = 0; stonesoup_i < stonesoupData->data_size; stonesoup_i++) {
        /* STONESOUP: TRIGGER-POINT (missinglockcheck) */
        if (stonesoupData->data[stonesoup_i] >= 'a' &&
            stonesoupData->data[stonesoup_i] <= 'z') {           /* null pointer
dereference possible */
            stonesoupData->data[stonesoup_i] -= 32;
        }
    }
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "TRIGGER-PONT: AFTER");
#endif /* STONESOUP_TRACE */

    pthread_mutex_unlock(&stonesoup_mutex);
    return NULL;
}

void *delNonAlpha (void *data) {
    struct stonesoup_data *stonesoupData = (struct stonesoup_data*) data;
    int i = 0;
    int j = 0;
    char* temp = malloc(sizeof(char) * (stonesoupData->data_size + 1));

#if STONESOUP_TRACE
```

```
    tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE,
"delNonAlpha");
#endif /* STONESOUP_TRACE */

    while(stonesoupData->data[i] != '\0') {
        if((stonesoupData->data[i] >= 'A' && stonesoupData->data[i] <= 'Z') ||
           (stonesoupData->data[i] >= 'a' && stonesoupData->data[i] <= 'z')) {
            temp[j++] = stonesoupData->data[i];
        }
        i++;
    }
    temp[j++] = '\0';

    stonesoupData->data_size = j;

    free(stonesoupData->data);

#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
    /* STONESOUP: CROSSOVER-POINT (missinglockcheck) */
    stonesoupData->data = NULL;                                  /* sets global ptr to
null, ka-boom */
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, variable_signed_integral, "stonesoupData->qsize",
stonesoupData->qsize, &(stonesoupData->qsize), "CROSSOVER-STATE");
    tracepoint(stonesoup_trace, variable_buffer, "stonesoupData->data",
stonesoupData->data, "CROSSOVER-STATE");
    tracepoint(stonesoup_trace, variable_buffer, "stonesoupData->file1",
stonesoupData->file1, "CROSSOVER-STATE");
    tracepoint(stonesoup_trace, variable_buffer, "stonesoupData->file2",
stonesoupData->file2, "CROSSOVER-STATE");
    tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
    stonesoup_readFile(stonesoupData->file2);

    stonesoupData->data = temp;
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, variable_signed_integral, "stonesoupData->qsize",
stonesoupData->qsize, &(stonesoupData->qsize), "POST CROSSOVER-STATE");
    tracepoint(stonesoup_trace, variable_buffer, "stonesoupData->data",
stonesoupData->data, "POST CROSSOVER-STATE");
    tracepoint(stonesoup_trace, variable_buffer, "stonesoupData->file1",
stonesoupData->file1, "POST CROSSOVER-STATE");
    tracepoint(stonesoup_trace, variable_buffer, "stonesoupData->file2",
stonesoupData->file2, "POST CROSSOVER-STATE");
#endif /* STONESOUP_TRACE */
    return NULL;
}

#endif /* STONESOUP_SNIPPET_SUPPORT */

#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
```

Approved for public release; distribution unlimited.

```
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 ******************************************************************************/
void weakness(char * stonesoup_taintedValue) {
#define STONESOUP_TAINT_SOURCE stonesoup_taintedValue
#endif /* STONESOUP_SNIPPET_DEBUG */

#if STONESOUP_SNIPPET_BODY_DECLARATIONS
    pthread_t stonesoup_t0, stonesoup_t1;
    struct stonesoup_data *stonesoupData;
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */

#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_start, "CWE414", "A", "Missing Lock Check");
#endif /* STONESOUP_TRACE */
    stonesoupData =  malloc(sizeof(struct stonesoup_data));
    if (stonesoupData) {
        stonesoupData->data  = malloc(sizeof(char) * (strlen(STONESOUP_TAINT_SOURCE) +
1));
        stonesoupData->file1 = malloc(sizeof(char) * (strlen(STONESOUP_TAINT_SOURCE) +
1));
        stonesoupData->file2 = malloc(sizeof(char) * (strlen(STONESOUP_TAINT_SOURCE) +
1));
        if (stonesoupData->data && stonesoupData->file1 && stonesoupData->file2) {
            if ((sscanf(STONESOUP_TAINT_SOURCE, "%d %s %s %s",
                      &(stonesoupData->qsize),
                        stonesoupData->file1,
                        stonesoupData->file2,
                        stonesoupData->data) == 4) &&
                (strlen(stonesoupData->data) != 0))
            {
#if STONESOUP_TRACE
                tracepoint(stonesoup_trace, variable_signed_integral,
"stonesoupData->qsize", stonesoupData->qsize, &(stonesoupData->qsize),
"INITIAL-STATE");
                tracepoint(stonesoup_trace, variable_buffer, "stonesoupData->data",
stonesoupData->data, "INITIAL-STATE");
                tracepoint(stonesoup_trace, variable_buffer, "stonesoupData->file1",
stonesoupData->file1, "INITIAL-STATE");
                tracepoint(stonesoup_trace, variable_buffer, "stonesoupData->file2",
stonesoupData->file2, "INITIAL-STATE");
#endif /* STONESOUP_TRACE */

                if (pthread_mutex_init(&stonesoup_mutex, NULL) != 0) {
                    stonesoup_printf("Mutex failed to initilize.");
                }
                stonesoupData->data_size = strlen(stonesoupData->data);

#if STONESOUP_TRACE
                tracepoint(stonesoup_trace, trace_point, "Spawning threads.");
#endif /* STONESOUP_TRACE */
                if (pthread_create(&stonesoup_t0, NULL, delNonAlpha, (void
*)stonesoupData) != 0) {  /* create thread that doesn't lock check */
                    stonesoup_printf("Error creating thread 0.");
                }

                if (pthread_create(&stonesoup_t1, NULL, toCap, (void *)stonesoupData)
```

```
!= 0) {
                    stonesoup_printf("Error creating thread 1.");
                }

                pthread_join(stonesoup_t0, NULL);
                pthread_join(stonesoup_t1, NULL);

#if STONESOUP_TRACE
                tracepoint(stonesoup_trace, trace_point, "Threads joined.");
#endif /* STONESOUP_TRACE */

                stonesoup_printf("After joins.\n");

                pthread_mutex_destroy(&stonesoup_mutex);
            } else {
                stonesoup_printf("Error parsing input.\n");
            }
            free(stonesoupData->data);
        }
        free(stonesoupData);
    }
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */

#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 ******************************************************************************/
int main(int argc,char *argv[])
{
    char* tainted_buff;
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    tainted_buff = argv[1];

    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
```

```
    return 0;
}
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-415A Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1
#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1
#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1
#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1
#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1
#endif /* NOTHING DEFINED */

#if STONESOUP_SNIPPET_INCLUDES
/*****************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 *****************************************************************************/
#include <stdlib.h> /* malloc */
#include <stdio.h> /* fprintf */
#include <string.h> /* memset */
#include <errno.h> /* errno */
/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_INCLUDES */
#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 *****************************************************************************/
```

```
#include "output-handlers/handler.h"
#endif /* STONESOUP_SNIPPET_DEBUG */
char stonesoup_process_buffer(char *buffer_param)
{
#if STONESOUP_TRACE
  tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE,
"stonesoup_process_buffer");
#endif /* STONESOUP_TRACE */
  char first_char;
  first_char = buffer_param[0] - 97;
  free(buffer_param);
  return first_char;
}
#endif /* STONESOUP_SNIPPET_SUPPORT */

#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 ******************************************************************************/
void weakness(char * stonesoup_tainted_buff) {
#define STONESOUP_TAINT_SOURCE stonesoup_tainted_buff
#endif /* STONESOUP_SNIPPET_DEBUG */
#if STONESOUP_SNIPPET_BODY_DECLARATIONS
 char *stonesoup_buffer = 0;
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_start, "CWE415", "A", "Double Free");
#endif /* STONESOUP_TRACE */
    stonesoup_buffer = malloc((strlen(STONESOUP_TAINT_SOURCE) + 1) * sizeof(char ));
    if (stonesoup_buffer == 0) {
        stonesoup_printf("Error: Failed to allocate memory\n");
        exit(1);
    }
    strcpy(stonesoup_buffer,STONESOUP_TAINT_SOURCE);
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
 /* STONESOUP: CROSSOVER-POINT (Double Free) */
    if (stonesoup_buffer[0] >= 97) {
        stonesoup_printf("Index of first char:
%i\n",stonesoup_process_buffer(stonesoup_buffer));
    }
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
    if (stonesoup_buffer != 0) {
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
  /* STONESOUP: TRIGGER-POINT (Double Free) */
        free(stonesoup_buffer);
```

```
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
    }
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 *****************************************************************************/
int main(int argc,char *argv[])
{
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    char* tainted_buff = argv[1];

    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
    return 0;
```

```
}
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-416A Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1
#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1
#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1
#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1
#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1
#endif /* NOTHING DEFINED */

#if STONESOUP_SNIPPET_INCLUDES
/*****************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 *****************************************************************************/
#include <stdlib.h> /* malloc */
#include <stdio.h> /* fprintf */
#include <string.h> /* memset */
#include <errno.h> /* errno */
/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_INCLUDES */
#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 *****************************************************************************/
#include "output-handlers/handler.h"
```

```
#endif /* STONESOUP_SNIPPET_DEBUG */
char stonesoup_process_buffer(char *buffer_param)
{
#if STONESOUP_TRACE
  tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE,
"stonesoup_process_buffer");
#endif /* STONESOUP_TRACE */
  char first_char;
  first_char = buffer_param[0] - 97;
  free(buffer_param);
  return first_char;
}
int stonesoup_toupper(int c)
{
  if (c >= 97 && c <= 122) {
    return c - 32;
  }
  return c;
}

#endif /* STONESOUP_SNIPPET_SUPPORT */

#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 *****************************************************************************/
void weakness(char * stonesoup_tainted_buff) {
#define STONESOUP_TAINT_SOURCE stonesoup_tainted_buff
#endif /* STONESOUP_SNIPPET_DEBUG */
#if STONESOUP_SNIPPET_BODY_DECLARATIONS
    int stonesoup_oc_i = 0;
    int stonesoup_tainted_len;
    char **stonesoup_buffer_ptr = 0;
    int stonesoup_main_first_char = 0;
    int stonesoup_buffer_len;
    char *stonesoup_buffer = 0;
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_start, "CWE416", "A", "Use After Free");
#endif /* STONESOUP_TRACE */
    stonesoup_buffer_len = 4;
    stonesoup_buffer = malloc(65528);
    if (stonesoup_buffer != NULL) {
        strncpy(stonesoup_buffer, STONESOUP_TAINT_SOURCE, stonesoup_buffer_len);
        stonesoup_buffer[stonesoup_buffer_len - 1] = 0;
        stonesoup_printf("%s\n", stonesoup_buffer);

#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, variable_signed_integral, "stonesoup_oc_i",
stonesoup_oc_i, &stonesoup_oc_i, "INITIAL-STATE");
        tracepoint(stonesoup_trace, variable_signed_integral, "stonesoup_tainted_len",
```

```
stonesoup_tainted_len, &stonesoup_tainted_len, "INITIAL-STATE");
        tracepoint(stonesoup_trace, variable_address, "stonesoup_buffer_ptr",
stonesoup_buffer_ptr, "INITIAL-STATE");
        tracepoint(stonesoup_trace, variable_signed_integral,
"stonesoup_main_first_char", stonesoup_main_first_char, &stonesoup_main_first_char,
"INITIAL-STATE");
        tracepoint(stonesoup_trace, variable_signed_integral, "stonesoup_buffer_len",
stonesoup_buffer_len, &stonesoup_buffer_len, "INITIAL-STATE");
        tracepoint(stonesoup_trace, variable_address, "stonesoup_buffer",
stonesoup_buffer, "INITIAL-STATE");
        tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
        /* STONESOUP: CROSSOVER-POINT (Use After Free) */
        if (stonesoup_buffer[0] >= 97) {
#if STONESOUP_TRACE
            tracepoint(stonesoup_trace, variable_address, "stonesoup_buffer",
stonesoup_buffer, "CROSSOVER-STATE");
#endif /* STONESOUP_TRACE */
            stonesoup_main_first_char = stonesoup_process_buffer(stonesoup_buffer);
        }
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
        stonesoup_buffer_ptr = malloc(65528);
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, variable_address, "stonesoup_buffer_ptr",
stonesoup_buffer_ptr, "CROSSOVER-STATE");
#endif /* STONESOUP_TRACE */
        if (stonesoup_buffer_ptr != NULL) {
            *stonesoup_buffer_ptr = STONESOUP_TAINT_SOURCE;
#if STONESOUP_TRACE
            tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
            tracepoint(stonesoup_trace, variable_address, "stonesoup_buffer",
stonesoup_buffer, "TRIGGER-STATE");
            tracepoint(stonesoup_trace, variable_address, "stonesoup_buffer_ptr",
stonesoup_buffer_ptr, "TRIGGER-STATE");
            tracepoint(stonesoup_trace, variable_address, "*stonesoup_buffer_ptr",
*stonesoup_buffer_ptr, "TRIGGER-STATE");
#endif /* STONESOUP_TRACE */
            /* STONESOUP: TRIGGER-POINT (Use After Free) */
            strncpy(stonesoup_buffer, STONESOUP_TAINT_SOURCE, stonesoup_buffer_len);
            stonesoup_buffer[stonesoup_buffer_len - 1] = '\0';
            stonesoup_tainted_len = strlen( *stonesoup_buffer_ptr); /* fail*/
#if STONESOUP_TRACE
            tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
            stonesoup_printf("%c\n", stonesoup_main_first_char);
            for (; stonesoup_oc_i < stonesoup_buffer_len; ++stonesoup_oc_i) {
                stonesoup_buffer[stonesoup_oc_i] =
stonesoup_toupper(stonesoup_buffer[stonesoup_oc_i]);
            }
            stonesoup_printf("%s\n", stonesoup_buffer);
            if (stonesoup_main_first_char == 0) {
                if (stonesoup_buffer != 0) {
                    free(stonesoup_buffer);
                }
            }
            if (stonesoup_buffer_ptr != 0) {
                free(stonesoup_buffer_ptr);
```

```
            }
        }
    }
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 *******************************************************************************/
int main(int argc,char *argv[])
{
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    char* tainted_buff = argv[1];

    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
    return 0;
```

```
}
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-476A Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1
#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1
#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1
#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1
#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1
#endif /* NOTHING DEFINED */

#if STONESOUP_SNIPPET_INCLUDES
/*****************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 *****************************************************************************/
#include <stdlib.h> // malloc
#include <stdio.h> // vfprintf
#include <string.h> // memcpy
/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_INCLUDES */
#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 *****************************************************************************/
#include "output-handlers/handler.h"
#endif /* STONESOUP_SNIPPET_DEBUG */
```

```
#endif /* STONESOUP_SNIPPET_SUPPORT */

#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 *****************************************************************************/
void weakness(char * stonesoup_tainted_buff) {
#define STONESOUP_TAINT_SOURCE stonesoup_tainted_buff
#endif /* STONESOUP_SNIPPET_DEBUG */
#define STONESOUP_EXPRESSION_1 "/opt/stonesoup/workspace/testData/myfile.txt"
#if STONESOUP_SNIPPET_BODY_DECLARATIONS
  FILE *stonesoup_csv = 0;
  FILE *stonesoup_temp = 0;
  char stonesoup_col1[80] = {0};
  char stonesoup_col2[80] = {0};
  char stonesoup_col3[80] = {0};
  char *stonesoup_cols[3] = {0};
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
  tracepoint(stonesoup_trace, weakness_start, "CWE476", "A", "NULL Pointer
Dereference");
  tracepoint(stonesoup_trace, variable_buffer, "STONESOUP_TAINT_SOURCE",
STONESOUP_TAINT_SOURCE, "INITIAL-STATE");
#endif /* STONESOUP_TRACE */
  stonesoup_csv = fopen(STONESOUP_TAINT_SOURCE,"r");
  if (stonesoup_csv != 0) {
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
/* STONESOUP: CROSSOVER-POINT (Syntactically Invalid Structure */

fscanf(stonesoup_csv,"\"%79[^\"]\",\"%79[^\"]\",\"%79[^\"]\"",stonesoup_col1,stonesoup
_col2,stonesoup_col3);
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, variable_buffer, "stonesoup_col1", stonesoup_col1,
"CROSSOVER-STATE");
    tracepoint(stonesoup_trace, variable_buffer, "stonesoup_col2", stonesoup_col2,
"CROSSOVER-STATE");
    tracepoint(stonesoup_trace, variable_buffer, "stonesoup_col3", stonesoup_col3,
"CROSSOVER-STATE");
    tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
    if (strlen(stonesoup_col1) > 0)
        stonesoup_cols[0] = stonesoup_col1;
    if (strlen(stonesoup_col2) > 0)
        stonesoup_cols[1] = stonesoup_col2;
    if (strlen(stonesoup_col3) > 0)
        stonesoup_cols[2] = stonesoup_col3;
    stonesoup_temp = fopen(STONESOUP_EXPRESSION_1, "w+");
    if(stonesoup_temp != 0) {
```

```
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
/* STONESOUP: TRIGGER-POINT (Syntactically Invalid Structure) */
        stonesoup_printf("VALUES=\"");
        fputs(stonesoup_cols[0],stonesoup_temp);
        stonesoup_printf(stonesoup_cols[0]);
        stonesoup_printf("\",\"");
        fputs(stonesoup_cols[1],stonesoup_temp);
        stonesoup_printf(stonesoup_cols[1]);
        stonesoup_printf("\",\"");
        fputs(stonesoup_cols[2],stonesoup_temp);
        stonesoup_printf(stonesoup_cols[2]);
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
        stonesoup_printf("\"\n");
        fclose(stonesoup_temp);
    }
  }
#if STONESOUP_TRACE
  tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 *******************************************************************************/
int main(int argc,char *argv[])
{
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    char* tainted_buff = argv[1];

    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
    exit(0);
}
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-476B Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1
#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1
#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1
#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1
#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1
#endif /* NOTHING DEFINED */

#if STONESOUP_SNIPPET_INCLUDES
/*******************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 *******************************************************************************/
#include <stdlib.h> // malloc
#include <stdio.h> // vfprintf
#include <string.h> // memcpy
/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_INCLUDES */
#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 *******************************************************************************/
#include "output-handlers/handler.h"
#endif /* STONESOUP_SNIPPET_DEBUG */
```

```
#endif /* STONESOUP_SNIPPET_SUPPORT */
#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 *****************************************************************************/
void weakness(char * stonesoup_tainted_buff) {
#define STONESOUP_TAINT_SOURCE stonesoup_tainted_buff
#endif /* STONESOUP_SNIPPET_DEBUG */
#if STONESOUP_SNIPPET_BODY_DECLARATIONS
  size_t stonesoup_j = 0;
  size_t stonesoup_i = 0;
  char *stonesoup_second_buff = 0;
  char *stonesoup_finder = "aba";
  int stonesoup_check = 0;
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
      tracepoint(stonesoup_trace, weakness_start, "CWE476", "B", "NULL Pointer
Dereference");
      tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
      for (stonesoup_i = 0; ((int )stonesoup_i) <= ((int
)(strlen(STONESOUP_TAINT_SOURCE) - strlen(stonesoup_finder))); ++stonesoup_i) {
        for (stonesoup_j = 0; stonesoup_j < strlen(stonesoup_finder); ++stonesoup_j) {
          if (STONESOUP_TAINT_SOURCE[stonesoup_i + stonesoup_j] !=
stonesoup_finder[stonesoup_j]) {
            stonesoup_check = 0;
            break;
          }
          stonesoup_check = 1;
        }
/* STONESOUP: CROSSOVER-POINT (Null Pointer Dereference) */
        if (stonesoup_check == 1 && stonesoup_j == strlen(stonesoup_finder)) {
          stonesoup_printf("Found aba string\n");
          stonesoup_second_buff = &STONESOUP_TAINT_SOURCE[stonesoup_i];
          break;
        }
      }
#if STONESOUP_TRACE
      tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
      tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
      tracepoint(stonesoup_trace, variable_address, "stonesoup_second_buff",
stonesoup_second_buff, "TRIGGER-STATE");
#endif /* STONESOUP_TRACE */
/* STONESOUP: TRIGGER-POINT (Null Pointer Dereference) */
      stonesoup_printf("String length is %i\n", strlen(stonesoup_second_buff));
#if STONESOUP_TRACE
      tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: AFTER");
      tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
```

```
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_DEBUG
/******************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 ******************************************************************************/
int main(int argc,char *argv[])
{
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    char* tainted_buff = argv[1];

    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
    exit(0);
}
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-476C Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1
#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1
#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1
#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1
#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1
#endif /* NOTHING DEFINED */

#if STONESOUP_SNIPPET_INCLUDES
/*****************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 *****************************************************************************/
#include <stdlib.h> // malloc
#include <stdio.h> // vfprintf
#include <string.h> // memcpy
/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_INCLUDES */
#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 *****************************************************************************/
#include "output-handlers/handler.h"
#endif /* STONESOUP_SNIPPET_DEBUG */
```

```
#endif /* STONESOUP_SNIPPET_SUPPORT */
#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 *****************************************************************************/
void weakness(char * stonesoup_tainted_buff) {
#define STONESOUP_TAINT_SOURCE stonesoup_tainted_buff
#endif /* STONESOUP_SNIPPET_DEBUG */
#define STONESOUP_EXPRESSION_1 "/opt/stonesoup/workspace/testData/myfile.txt"
#if STONESOUP_SNIPPET_BODY_DECLARATIONS
  FILE *stonesoup_temp = 0;
  int stonesoup_i;
  char **stonesoup_values;
  int stonesoup_len;
  char stonesoup_temp_str[80];
  char *stonesoup_endptr;
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
      tracepoint(stonesoup_trace, weakness_start, "CWE476", "C", "NULL Pointer
Dereference");
#endif /* STONESOUP_TRACE */
      stonesoup_len = strtol(STONESOUP_TAINT_SOURCE,&stonesoup_endptr,10);
      if (stonesoup_len > 0 && stonesoup_len < 1000) {
        stonesoup_values = malloc(stonesoup_len * sizeof(char *));
        if (stonesoup_values == 0) {
          stonesoup_printf("Error: Failed to allocate memory\n");
          exit(1);
        }
        for (stonesoup_i = 0; stonesoup_i < stonesoup_len; ++stonesoup_i)
          stonesoup_values[stonesoup_i] = 0;
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
        for (stonesoup_i = 0; stonesoup_i < stonesoup_len; ++stonesoup_i) {
/* STONESOUP: CROSSOVER-POINT (Null Pointer Dereference) */
          if (sscanf(stonesoup_endptr," %79s",stonesoup_temp_str) == 1) {
            stonesoup_values[stonesoup_i] = ((char
*)(malloc((strlen(stonesoup_temp_str) + 1) * sizeof(char ))));
            if (stonesoup_values[stonesoup_i] == 0) {
              stonesoup_printf("Error: Failed to allocate memory\n");
              exit(1);
            }
            strcpy(stonesoup_values[stonesoup_i],stonesoup_temp_str);
            stonesoup_endptr += (strlen(stonesoup_temp_str) + 1) * sizeof(char );
          }
        }
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
        stonesoup_temp = fopen(STONESOUP_EXPRESSION_1, "w+");
        if(stonesoup_temp != 0) {
```

```
#if STONESOUP_TRACE
          tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
          tracepoint(stonesoup_trace, variable_signed_integral, "stonesoup_len",
stonesoup_len, &stonesoup_len, "TRIGGER-STATE");
#endif /* STONESOUP_TRACE */
          for (stonesoup_i = 0; stonesoup_i < stonesoup_len; ++stonesoup_i) {
/* STONESOUP: TRIGGER-POINT (Null Pointer Dereference) */
#if STONESOUP_TRACE
            tracepoint(stonesoup_trace, variable_buffer,
"stonesoup_values[stonesoup_i]", stonesoup_values[stonesoup_i], "TRIGGER-STATE");
#endif /* STONESOUP_TRACE */
            fputs(stonesoup_values[stonesoup_i],stonesoup_temp);
            stonesoup_printf(stonesoup_values[stonesoup_i]);
          }
#if STONESOUP_TRACE
          tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
          fclose(stonesoup_temp);
        }
        stonesoup_printf("\n");
        for (stonesoup_i = 0; stonesoup_i < stonesoup_len; ++stonesoup_i)
          if (stonesoup_values[stonesoup_i] != 0) {
            free(stonesoup_values[stonesoup_i]);
          }
        if (stonesoup_values != 0) {
          free(stonesoup_values);
        }
      }
#if STONESOUP_TRACE
      tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 *****************************************************************************/
int main(int argc,char *argv[])
{
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    char* tainted_buff = argv[1];

    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
    exit(0);
}
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-476D Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1
#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1
#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1
#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1
#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1
#endif /* NOTHING DEFINED */

#if STONESOUP_SNIPPET_INCLUDES
/******************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 ******************************************************************************/
#include <stdlib.h> // malloc
#include <stdio.h> // vfprintf
#include <string.h> // memcpy
/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_INCLUDES */
#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
/******************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 ******************************************************************************/
#include "output-handlers/handler.h"
#endif /* STONESOUP_SNIPPET_DEBUG */
```

```
int stonesoup_476_global_variable = 0;
int stonesoup_isalnum(int c)
{
  if ((c >= 97 && c <= 122) || (c >= 65 && c <= 90) || (c >= 48 && c <= 57)) {
    return 1;
  }
  return 0;
}
#endif /* STONESOUP_SNIPPET_SUPPORT */
#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 *****************************************************************************/
void weakness(char * stonesoup_tainted_buff) {
#define STONESOUP_TAINT_SOURCE stonesoup_tainted_buff
#endif /* STONESOUP_SNIPPET_DEBUG */
#if STONESOUP_SNIPPET_BODY_DECLARATIONS
  char *stonesoup_second_buff = 0;
  int stonesoup_size = 0;
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
      tracepoint(stonesoup_trace, weakness_start, "CWE476", "D", "NULL Pointer
Dereference");
#endif /* STONESOUP_TRACE */
      while(stonesoup_isalnum(STONESOUP_TAINT_SOURCE[stonesoup_size]) &&
stonesoup_size < strlen(STONESOUP_TAINT_SOURCE)){
        ++stonesoup_size;
      }
#if STONESOUP_TRACE
      tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
/* STONESOUP: CROSSOVER-POINT (Null Pointer Dereference) */
      if (stonesoup_size != strlen(STONESOUP_TAINT_SOURCE)) {
        STONESOUP_TAINT_SOURCE = 0;
      }
#if STONESOUP_TRACE
      tracepoint(stonesoup_trace, variable_address, "STONESOUP_TAINT_SOURCE",
STONESOUP_TAINT_SOURCE, "CROSSOVER-STATE");
      tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
      stonesoup_second_buff = malloc((stonesoup_size + 1) * sizeof(char ));
      if (stonesoup_second_buff == 0) {
        stonesoup_printf("Error: Failed to allocate memory\n");
        exit(1);
      }
#if STONESOUP_TRACE
      tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
/* STONESOUP: TRIGGER-POINT (Null Pointer Dereference) */
      strcpy(stonesoup_second_buff,STONESOUP_TAINT_SOURCE);
      stonesoup_476_global_variable = strlen(stonesoup_second_buff);
```

```
#if STONESOUP_TRACE
      tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
      ++stonesoup_476_global_variable;
      if (stonesoup_second_buff != 0) {
        free(stonesoup_second_buff);
      }

      stonesoup_printf("String contains only alpha numeric characters\n");
#if STONESOUP_TRACE
      tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 *******************************************************************************/
int main(int argc,char *argv[])
{
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    char* tainted_buff = argv[1];

    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
    exit(0);
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1
#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1
#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1
#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1
#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1
#endif /* NOTHING DEFINED */

#if STONESOUP_SNIPPET_INCLUDES
/*******************************************************************************
 * Weakness Snippet Dependencies
```

Approved for public release; distribution unlimited.

```
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 *****************************************************************************/
#include <stdlib.h> // malloc
#include <stdio.h> // vfprintf
#include <string.h> // memcpy
/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_INCLUDES */
#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 *****************************************************************************/
#include "output-handlers/handler.h"
#endif /* STONESOUP_SNIPPET_DEBUG */
int stonesoup_476_global_variable = 0;
int stonesoup_isalnum(int c)
{
  if ((c >= 97 && c <= 122) || (c >= 65 && c <= 90) || (c >= 48 && c <= 57)) {
    return 1;
  }
  return 0;
}
#endif /* STONESOUP_SNIPPET_SUPPORT */
#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 *****************************************************************************/
void weakness(char * stonesoup_tainted_buff) {
#define STONESOUP_TAINT_SOURCE stonesoup_tainted_buff
#endif /* STONESOUP_SNIPPET_DEBUG */
#if STONESOUP_SNIPPET_BODY_DECLARATIONS
  char *stonesoup_second_buff = 0;
  int stonesoup_size = 0;
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_BODY_STATEMENTS
```

```
#if STONESOUP_TRACE
      tracepoint(stonesoup_trace, weakness_start, "CWE476", "D", "NULL Pointer
Dereference");
#endif /* STONESOUP_TRACE */
      while(stonesoup_isalnum(STONESOUP_TAINT_SOURCE[stonesoup_size]) &&
stonesoup_size < strlen(STONESOUP_TAINT_SOURCE)){
        ++stonesoup_size;
      }
#if STONESOUP_TRACE
      tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
/* STONESOUP: CROSSOVER-POINT (Null Pointer Dereference) */
      if (stonesoup_size != strlen(STONESOUP_TAINT_SOURCE)) {
        STONESOUP_TAINT_SOURCE = 0;
      }
#if STONESOUP_TRACE
      tracepoint(stonesoup_trace, variable_address, "STONESOUP_TAINT_SOURCE",
STONESOUP_TAINT_SOURCE, "CROSSOVER-STATE");
      tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
      stonesoup_second_buff = malloc((stonesoup_size + 1) * sizeof(char ));
      if (stonesoup_second_buff == 0) {
        stonesoup_printf("Error: Failed to allocate memory\n");
        exit(1);
      }
#if STONESOUP_TRACE
      tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
/* STONESOUP: TRIGGER-POINT (Null Pointer Dereference) */
      strcpy(stonesoup_second_buff,STONESOUP_TAINT_SOURCE);
      stonesoup_476_global_variable = strlen(stonesoup_second_buff);
#if STONESOUP_TRACE
      tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
      ++stonesoup_476_global_variable;
      if (stonesoup_second_buff != 0) {
        free(stonesoup_second_buff);
      }

      stonesoup_printf("String contains only alpha numeric characters\n");
#if STONESOUP_TRACE
      tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_DEBUG
/******************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 ******************************************************************************/
int main(int argc,char *argv[])
{
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
```

```
    }
    char* tainted_buff = argv[1];

    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
    exit(0);
}
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-476E Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1
#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1
#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1
#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1
#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1
#endif /* NOTHING DEFINED */

#if STONESOUP_SNIPPET_INCLUDES
/*****************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 *****************************************************************************/
#include <stdlib.h> // malloc
#include <stdio.h> // vfprintf
#include <string.h> // memcpy
/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_INCLUDES */
#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 *****************************************************************************/
#include "output-handlers/handler.h"
#endif /* STONESOUP_SNIPPET_DEBUG */
```

```
#endif /* STONESOUP_SNIPPET_SUPPORT */
#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 ******************************************************************************/
void weakness(char * stonesoup_tainted_buff) {
#define STONESOUP_TAINT_SOURCE stonesoup_tainted_buff
#endif /* STONESOUP_SNIPPET_DEBUG */
#if STONESOUP_SNIPPET_BODY_DECLARATIONS
  char stonesoup_buffer[80];
  FILE *stonesoup_pFile = 0;
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
      tracepoint(stonesoup_trace, weakness_start, "CWE476", "E", "NULL Pointer
Dereference");
      tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
/* STONESOUP: CROSSOVER-POINT */
      stonesoup_pFile = fopen(STONESOUP_TAINT_SOURCE,"r");
      stonesoup_buffer[0] = 0;
#if STONESOUP_TRACE
      tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
      tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
      tracepoint(stonesoup_trace, variable_address, "stonesoup_pFile",
stonesoup_pFile, "TRIGGER-STATE");
#endif /* STONESOUP_TRACE */
/* STONESOUP: TRIGGER-POINT (Null Pointer Dereference: Unchecked file read) */
      fgets(stonesoup_buffer,79,stonesoup_pFile);
      stonesoup_printf(stonesoup_buffer);
      stonesoup_printf("\n");
      fclose(stonesoup_pFile);
#if STONESOUP_TRACE
      tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: AFTER");
      tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 ******************************************************************************/
int main(int argc,char *argv[])
{
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
```

```
    }
    char* tainted_buff = argv[1];

    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
    exit(0);
}
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-476F Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1
#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1
#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1
#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1
#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1
#endif /* NOTHING DEFINED */

#if STONESOUP_SNIPPET_INCLUDES
/*******************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 *******************************************************************************/
#include <stdlib.h> // malloc
#include <stdio.h> // vfprintf
#include <string.h> // memcpy
/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_INCLUDES */
#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 *******************************************************************************/
#include "output-handlers/handler.h"
#endif /* STONESOUP_SNIPPET_DEBUG */
```

```
void *my_malloc(unsigned int size)
{
  if (size > 512)
/* STONESOUP: CROSSOVER-POINT */
    return 0;
  return malloc(size);
}
#endif /* STONESOUP_SNIPPET_SUPPORT */
#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 *****************************************************************************/
void weakness(char * stonesoup_tainted_buff) {
#define STONESOUP_TAINT_SOURCE stonesoup_tainted_buff
#endif /* STONESOUP_SNIPPET_DEBUG */
#if STONESOUP_SNIPPET_BODY_DECLARATIONS
  unsigned int stonesoup_size_buffer;
  int stonesoup_buffer_value;
  char *stonesoup_malloc_buffer = 0;
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
      tracepoint(stonesoup_trace, weakness_start, "CWE476", "F", "NULL Pointer
Dereference");
#endif /* STONESOUP_TRACE */
      stonesoup_buffer_value = atoi(STONESOUP_TAINT_SOURCE);
#if STONESOUP_TRACE
      tracepoint(stonesoup_trace, variable_signed_integral, "stonesoup_buffer_value",
stonesoup_buffer_value, &stonesoup_buffer_value, "INITIAL-STATE");
#endif /* STONESOUP_TRACE */
      if (stonesoup_buffer_value < 0)
        stonesoup_buffer_value = 0;
      stonesoup_size_buffer = ((unsigned int )stonesoup_buffer_value);
#if STONESOUP_TRACE
      tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
      stonesoup_malloc_buffer = my_malloc(stonesoup_size_buffer);
#if STONESOUP_TRACE
      tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
      tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
      tracepoint(stonesoup_trace, variable_address, "stonesoup_malloc_buffer",
stonesoup_malloc_buffer, "TRIGGER-STATE");
#endif /* STONESOUP_TRACE */
/* STONESOUP: TRIGGER-POINT (Null Pointer Dereference: Wrapped malloc) */
      memset(stonesoup_malloc_buffer,0,stonesoup_size_buffer);
      stonesoup_printf("Buffer size is %d\n", stonesoup_size_buffer);
#if STONESOUP_TRACE
      tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
      if (stonesoup_malloc_buffer != 0) {
        free(stonesoup_malloc_buffer);
      }
```

```
#if STONESOUP_TRACE
      tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 ******************************************************************************/
int main(int argc,char *argv[])
{
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    char* tainted_buff = argv[1];

    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
    exit(0);
}
#endif /* STONESOUP_SNIPPET_DEBUG */
```

Approved for public release; distribution unlimited.

C - CWE-476G Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1
#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1
#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1
#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1
#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1
#endif /* NOTHING DEFINED */

#if STONESOUP_SNIPPET_INCLUDES
/*****************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 *****************************************************************************/
#include <stdlib.h> // malloc
#include <stdio.h> // vfprintf
#include <string.h> // memcpy
/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_INCLUDES */
#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 *****************************************************************************/
#include "output-handlers/handler.h"
#endif /* STONESOUP_SNIPPET_DEBUG */
```

```
#endif /* STONESOUP_SNIPPET_SUPPORT */
#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/******************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 ******************************************************************************/
void weakness(char * stonesoup_tainted_buff) {
#define STONESOUP_TAINT_SOURCE stonesoup_tainted_buff
#endif /* STONESOUP_SNIPPET_DEBUG */
#if STONESOUP_SNIPPET_BODY_DECLARATIONS
  char *stonesoup_skip_malloc_buffer = 0;
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
      tracepoint(stonesoup_trace, weakness_start, "CWE476", "G", "NULL Pointer
Dereference");
      tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
/* STONESOUP: CROSSOVER-POINT */
      if (strlen(STONESOUP_TAINT_SOURCE) < 63) {
        stonesoup_skip_malloc_buffer = malloc(strlen(STONESOUP_TAINT_SOURCE + 1));
      }
#if STONESOUP_TRACE
      tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
      tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
      tracepoint(stonesoup_trace, variable_address, "stonesoup_skip_malloc_buffer",
stonesoup_skip_malloc_buffer, "TRIGGER-STATE");
#endif /* STONESOUP_TRACE */
/* STONESOUP: TRIGGER-POINT (Null Pointer Dereference: Unchecked strcpy) */
      strcpy(stonesoup_skip_malloc_buffer,STONESOUP_TAINT_SOURCE);
      stonesoup_printf("Buffer is %s\n",stonesoup_skip_malloc_buffer);
#if STONESOUP_TRACE
      tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
      if (stonesoup_skip_malloc_buffer != 0) {
        free(stonesoup_skip_malloc_buffer);
      }
#if STONESOUP_TRACE
      tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_DEBUG
/******************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 ******************************************************************************/
int main(int argc,char *argv[])
{
```

```
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    char* tainted_buff = argv[1];

    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
    exit(0);
}
#endif /* STONESOUP_SNIPPET_DEBUG */
```

Approved for public release; distribution unlimited.

C - CWE-479A Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1

#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1

#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1

#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1

#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1

#endif /* NOTHING DEFINED */


#if STONESOUP_SNIPPET_INCLUDES
/******************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 ******************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <math.h>
#include <pthread.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>

/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */

#endif /* STONESOUP_SNIPPET_INCLUDES */
```

```
#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 ******************************************************************************/
#include "output-handlers/handler.h"

#endif /* STONESOUP_SNIPPET_DEBUG */

struct stonesoup_data {
    int data_size;
    char *data;
    char *file1;
};

struct stonesoup_data *stonesoupData;
pthread_mutex_t stonesoup_mutex;

int stonesoup_comp (const void * a, const void * b)
{
    if (a > b) {
        return -1;
    }
    else if (a < b) {
        return 1;
    }
    else {
        return 0;
    }
}

int stonesoup_pmoc (const void * a, const void * b)
{
    return -1 * stonesoup_comp(a, b);
}

void stonesoup_readFile(char *filename) {
    FILE *fifo;
    char ch;

#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE,
"stonesoup_readFile");
#endif /* STONESOUP_TRACE */

    fifo = fopen(filename, "r");

    if (fifo != NULL) {
        while ((ch = fgetc(fifo)) != EOF) {
            stonesoup_printf("%c", ch);
        }
```

```
        fclose(fifo);
    }
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "Finished reading sync file");
#endif /* STONESOUP_TRACE */
}

void waitForSig(char* sleepFile) {
    int fd;
    char outStr[25] = {0};
    char filename[500] = {0};

#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE,
"waitForSig");
#endif /* STONESOUP_TRACE */

    stonesoup_printf("In waitForSig\n");

    sprintf(outStr, "%d.pid", getpid());
    strcat(filename, STONESOUP_TESTDATA);
    strcat(filename, outStr);

    if ((fd = open(filename, O_CREAT|O_WRONLY, 0666)) == -1) {
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, trace_error, "Error opening file.");
#endif /* STONESOUP_TRACE */
        stonesoup_printf("Error opening file.");
    }
    else {
        if (write(fd, "q", sizeof(char)) == -1) {
            stonesoup_printf("Error writing to file.");
        }

#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, trace_point, "Wrote .pid file.");
#endif /* STONESOUP_TRACE */

        if (close(fd) == -1) {
            stonesoup_printf("Error closing file.");
        }
        stonesoup_readFile(sleepFile);
    }
}

void delNonAlpha (void *data) {
    struct stonesoup_data *stonesoupData = (struct stonesoup_data*) data;
    int i = 0;
    int j = 0;
    char* temp = malloc(sizeof(char) * (stonesoupData->data_size + 1));

#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE,
"delNonAlpha");
#endif /* STONESOUP_TRACE */

    stonesoup_printf("Grabbing lock\n");

#if STONESOUP_TRACE
```

```
    tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
    /* STONESOUP: TRIGGER-POINT (nonreentrant signal handler) */
    pthread_mutex_lock(&stonesoup_mutex);       /* mutex lock causes deadlock on
re-entrance */
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "mutex locked");
    tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: AFTER");
#endif /* STONESOUP_TRACE */

    while(stonesoupData->data[i] != '\0') {
        if((stonesoupData->data[i] >= 'A' && stonesoupData->data[i] <= 'Z') ||
           (stonesoupData->data[i] >= 'a' && stonesoupData->data[i] <= 'z')) {
            temp[j++] = stonesoupData->data[i];
        }
        i++;
    }
    temp[j++] = '\0';

    stonesoupData->data_size = j;
    free(stonesoupData->data);
    stonesoupData->data = temp;

    waitForSig(stonesoupData->file1);       /* Deadlock */

    stonesoup_printf("Realeasing lock\n");
    pthread_mutex_unlock(&stonesoup_mutex);
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "mutex unlocked");
#endif /* STONESOUP_TRACE */
}

void sig_handler (int sig) {
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE,
"sig_handler");
#endif /* STONESOUP_TRACE */
    /* STONESOUP: CROSSOVER-POINT (nonreentrentsighandler) */
    if (stonesoupData != NULL) {
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
        tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
        delNonAlpha(stonesoupData);         /* call non-reentrant function - deadlock
*/
    }

    signal(SIGUSR1, SIG_IGN);
}

#endif /* STONESOUP_SNIPPET_SUPPORT */

#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
```

```
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 ******************************************************************************/
void weakness(char * taintedValue) {
#define STONESOUP_TAINT_SOURCE taintedValue
#endif /* STONESOUP_SNIPPET_DEBUG */
#if STONESOUP_SNIPPET_BODY_DECLARATIONS
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_start, "CWE479", "A", "Signal Handler Use of
a Non-reentrant Function");
#endif /* STONESOUP_TRACE */
    stonesoupData =  malloc(sizeof(struct stonesoup_data));
    if (stonesoupData) {
        stonesoupData->data  = malloc(sizeof(char) * (strlen(STONESOUP_TAINT_SOURCE) +
1));
        stonesoupData->file1 = malloc(sizeof(char) * (strlen(STONESOUP_TAINT_SOURCE) +
1));
        if (stonesoupData->data && stonesoupData->file1) {
            if ((sscanf(STONESOUP_TAINT_SOURCE, "%s %s",
                        stonesoupData->file1,
                        stonesoupData->data)  == 2) &&
                (strlen(stonesoupData->data)  != 0) &&
                (strlen(stonesoupData->file1) != 0))
            {
#if STONESOUP_TRACE
                tracepoint(stonesoup_trace, variable_buffer, "stonesoupData->data",
stonesoupData->data, "INITIAL-STATE");
                tracepoint(stonesoup_trace, variable_buffer, "stonesoupData->file1",
stonesoupData->file1, "INITIAL-STATE");
#endif /* STONESOUP_TRACE */
                stonesoupData->data_size = strlen(stonesoupData->data);
                if (pthread_mutex_init(&stonesoup_mutex, NULL) != 0) {
                    stonesoup_printf("Mutex failed to initilize.");
                 }
                if (signal(SIGUSR1, sig_handler) == SIG_ERR) {       /* setup signal
handler */
#if STONESOUP_TRACE
                    tracepoint(stonesoup_trace, trace_error, "Error setting up sig
handler for SIGUSR1");
#endif /* STONESOUP_TRACE */
                    stonesoup_printf ("Error catching SIGUSR1!\n");
                }
                delNonAlpha(stonesoupData);
                signal(SIGUSR1, SIG_IGN);                           /* 'deregister'
signal hander befor returning to base program */
            } else {
#if STONESOUP_TRACE
                tracepoint(stonesoup_trace, trace_error, "Error parsing data");
#endif /* STONESOUP_TRACE */
                stonesoup_printf("Error parsing data\n");
            }
            free(stonesoupData->data);
        }
        free(stonesoupData);
    }
#if STONESOUP_TRACE
```

```
    tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */

#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 ******************************************************************************/
int main(int argc,char *argv[])
{
    char* tainted_buff;
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    tainted_buff = argv[1];

    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
```

```
    return 0;
}
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-543A Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1

#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1

#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1

#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1

#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1

#endif /* NOTHING DEFINED */


#if STONESOUP_SNIPPET_INCLUDES
/*****************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 *****************************************************************************/
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <string.h>

/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */

#endif /* STONESOUP_SNIPPET_INCLUDES */

#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
```

```
/*******************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 ******************************************************************************/
#include "output-handlers/handler.h"

#endif /* STONESOUP_SNIPPET_DEBUG */

struct stonesoup_list {
    int data;
    struct stonesoup_list *previous;
    struct stonesoup_list *next;
};

struct stonesoup_queue {
    pthread_mutex_t lock;
    pthread_cond_t is_empty;
    pthread_cond_t is_full;
    int size;
    int capacity;
    struct stonesoup_list *head;
    struct stonesoup_list *tail;
};

struct stonesoup_data {
    int qsize;
    int data;
    char* file1;
    char* file2;
};

int stonesoup_comp (const void * a, const void * b)
{
    if (a > b) {
        return -1;
    }
    else if (a < b) {
        return 1;
    }
    else {
        return 0;
    }
}

int stonesoup_pmoc (const void * a, const void * b)
{
    return -1 * stonesoup_comp(a, b);
}

void stonesoup_readFile(char *filename) {
    FILE *fifo;
    char ch;

#if STONESOUP_TRACE
```

```
    tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE,
"stonesoup_readFile");
#endif /* STONESOUP_TRACE */

    fifo = fopen(filename, "r");

    if (fifo != NULL) {
        while ((ch = fgetc(fifo)) != EOF) {
            stonesoup_printf("%c", ch);
        }

        fclose(fifo);
    }
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE, "Finished
reading sync file.");
#endif /* STONESOUP_TRACE */
}

int enqueue(struct stonesoup_queue *ssQ, int data) {
    int rtnval = 0;

    if (ssQ != NULL) {
        struct stonesoup_list *elem = malloc(sizeof(struct stonesoup_list));
        pthread_mutex_lock(&(ssQ->lock));
        while (ssQ->size >= ssQ->capacity) {
            pthread_cond_wait(&(ssQ->is_full), &(ssQ->lock));
        }
        elem->next = NULL;
        elem->previous = ssQ->tail;
        elem->data = data;
        if (ssQ->tail != NULL) {
            ssQ->tail->next = elem;
        }
        ssQ->tail = elem;
        ssQ->size++;
        if (ssQ->head == NULL) {
            ssQ->head = elem;
        }
        pthread_mutex_unlock(&(ssQ->lock));
        pthread_cond_broadcast(&(ssQ->is_empty));
        }
    else {
        rtnval = -1;
    }

    return rtnval;
}

int dequeue(struct stonesoup_queue *ssQ) {
    int val = -1;

    if (ssQ != NULL) {
        struct stonesoup_list *elem;
        pthread_mutex_lock(&(ssQ->lock));
        while (ssQ->size <= 0) {
            pthread_cond_wait(&(ssQ->is_empty), &(ssQ->lock));
        }
        elem = ssQ->head;
```

```
        ssQ->head = elem->next;
        if(ssQ->head != NULL) {
            ssQ->head->previous = NULL;
        }
        else {
            ssQ->tail = NULL;
        }
        val = elem->data;
        ssQ->size--;
        free(elem);
        pthread_mutex_unlock(&(ssQ->lock));
        pthread_cond_broadcast(&(ssQ->is_full));
    }
    return val;
}

struct stonesoup_queue *get_instance (char* file2) {
    static struct stonesoup_queue *ssQ = NULL;

#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE,
"get_instance");
    tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */

    /* STONESOUP: CROSSOVER-POINT (singletonpatternwithoutsync) */
    if (ssQ == NULL) {
        if (file2 != NULL) {
            stonesoup_readFile(file2);
        }

        ssQ = (struct stonesoup_queue *)calloc(1, sizeof(struct stonesoup_queue));

        pthread_mutex_init(&(ssQ->lock), NULL);
        pthread_cond_init(&(ssQ->is_empty), NULL);
        pthread_cond_init(&(ssQ->is_full), NULL);
        ssQ->size = 0;
        ssQ->capacity = 30;
        ssQ->head = NULL;
        ssQ->tail = NULL;
    }

#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
    tracepoint(stonesoup_trace, variable_address, "ssQ", ssQ, "CROSSOVER-STATE");
#endif /* STONESOUP_TRACE */

    return ssQ;
}

void *stonesoup_print_data (void *data) {
    struct stonesoup_data *ssD = (struct stonesoup_data *)data;
    struct stonesoup_queue *ssQ = get_instance(ssD->file2);
    int i;

#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE,
"stonesoup_print_data");
    tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
```

```
#endif /* STONESOUP_TRACE */

    /* STONESOUP: TRIGGER-POINT (singletonpatternwithoutsync) */
    while ((i = dequeue(ssQ)) != -1) {
        stonesoup_printf("Data: %d\n", i);
    }

#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: AFTER");
#endif /* STONESOUP_TRACE */

    return NULL;
}

void *stonesoup_calc_data (void *data) {
    struct stonesoup_data *ssD = (struct stonesoup_data *)data;
    struct stonesoup_queue *ssQ;
    int *qsort_arr;
    int i;

#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE,
"stonesoup_calc_data");
#endif /* STONESOUP_TRACE */

    qsort_arr = malloc(sizeof(int)*ssD->qsize);
        if (qsort_arr != NULL) {
            for (i = 0; i < ssD->qsize; i++) {
                qsort_arr[i] = ssD->qsize - i;
            }
            qsort(qsort_arr, ssD->qsize, sizeof(int), &stonesoup_comp);
            free (qsort_arr);
            qsort_arr = NULL;
        }

    stonesoup_readFile(ssD->file1);

    ssQ = get_instance(NULL);

#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT2: BEFORE");
#endif /* STONESOUP_TRACE */
    for (i = 0; i < ssD->data; i++) {
        /* STONESOUP2: TRIGGER-POINT (singletonpatternwithoutsync) */
        if (enqueue(ssQ, i) == -1) {
            break;
        }
    }
    enqueue(ssQ, -1);
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT2: AFTER");
#endif /* STONESOUP_TRACE */

    return NULL;
}

#endif /* STONESOUP_SNIPPET_SUPPORT */

#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
```

```
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 *****************************************************************************/
void weakness(char * stonesoup_taintedValue) {
#define STONESOUP_TAINT_SOURCE stonesoup_taintedValue
#endif /* STONESOUP_SNIPPET_DEBUG */

#if STONESOUP_SNIPPET_BODY_DECLARATIONS
    pthread_t stonesoup_t0, stonesoup_t1;
    struct stonesoup_data* stonesoupData;
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */

#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_start, "CWE543", "A", "Use of a Singleton
Pattern Without Synchronization in a Multithreaded Context");
#endif /* STONESOUP_TRACE */

    stonesoupData =  malloc(sizeof(struct stonesoup_data));
    if (stonesoupData) {
        stonesoupData->file1 = malloc(sizeof(char) * (strlen(STONESOUP_TAINT_SOURCE) +
1));
        stonesoupData->file2 = malloc(sizeof(char) * (strlen(STONESOUP_TAINT_SOURCE) +
1));
        if ((sscanf(STONESOUP_TAINT_SOURCE, "%d %s %s %d",
                  &(stonesoupData->qsize),
                    stonesoupData->file1,
                    stonesoupData->file2,
                  &(stonesoupData->data)) == 4) &&
                    stonesoupData->qsize  >= 0  &&
                    stonesoupData->data   >= 0  &&
            (strlen(stonesoupData->file1) != 0) &&
            (strlen(stonesoupData->file2) != 0))
        {
#if STONESOUP_TRACE
            tracepoint(stonesoup_trace, variable_signed_integral,
"stonesoupData->qsize", stonesoupData->qsize, &(stonesoupData->qsize),
"INITIAL-STATE");
            tracepoint(stonesoup_trace, variable_signed_integral,
"stonesoupData->data", stonesoupData->data, &(stonesoupData->data), "INITIAL-STATE");
            tracepoint(stonesoup_trace, variable_buffer, "stonesoupData->file1",
stonesoupData->file1, "INITIAL-STATE");
            tracepoint(stonesoup_trace, variable_buffer, "stonesoupData->file2",
stonesoupData->file2, "INITIAL-STATE");
            tracepoint(stonesoup_trace, trace_point, "Spawning threads.");
#endif /* STONESOUP_TRACE */

            if (pthread_create(&stonesoup_t0, NULL, stonesoup_calc_data,
stonesoupData) != 0) {
                stonesoup_printf("Thread 0 failed to spawn.");
            }
            if (pthread_create(&stonesoup_t1, NULL, stonesoup_print_data,
```

```
stonesoupData) != 0) {
                stonesoup_printf("Thread 1 failed to spawn.");
            }

            pthread_join(stonesoup_t0, NULL);
            pthread_join(stonesoup_t1, NULL);
#if STONESOUP_TRACE
            tracepoint(stonesoup_trace, trace_point, "Threads joined.");
#endif /* STONESOUP_TRACE */
        } else {
#if STONESOUP_TRACE
            tracepoint(stonesoup_trace, trace_error, "Error parsng data.");
#endif /* STONESOUP_TRACE */
            stonesoup_printf("Error parsing data\n");
        }
        free(stonesoupData->file1);
        free(stonesoupData->file2);
        free(stonesoupData);
    }
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */

#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 *******************************************************************************/
int main(int argc,char *argv[])
{
    char* tainted_buff;
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    tainted_buff = argv[1];

    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
```

```
    return 0;
}
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-590A Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1
#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1
#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1
#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1
#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1
#endif /* NOTHING DEFINED */

#if STONESOUP_SNIPPET_INCLUDES
/*******************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 *******************************************************************************/
#include <stdlib.h> /* malloc */
#include <stdio.h> /* fprintf */
#include <string.h> /* memset */
#include <errno.h> /* errno */
/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_INCLUDES */
#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 *******************************************************************************/
```

```
#include "output-handlers/handler.h"
#endif /* STONESOUP_SNIPPET_DEBUG */
int stonesoup_toupper(int c) {
  if (c >= 97 && c <= 122) {
    return c - 32;
  }
  return c;
}
void stonesoup_free_func(char *buff) {
#if STONESOUP_TRACE
  tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE,
"stonesoup_free_func");
#endif /* STONESOUP_TRACE */
  if (buff[0] >= 79) {
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
    tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
    /* STONESOUP: CROSSOVER-POINT (Free of Invalid Pointer Not on the Heap) */
    /* STONESOUP: TRIGGER-POINT (Free of Invalid Pointer Not on the Heap) */
    free(buff);
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: AFTER");
    tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
  }
}
#endif /* STONESOUP_SNIPPET_SUPPORT */

#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 *****************************************************************************/
void weakness(char * stonesoup_tainted_buff) {
#define STONESOUP_TAINT_SOURCE stonesoup_tainted_buff
#endif /* STONESOUP_SNIPPET_DEBUG */
#if STONESOUP_SNIPPET_BODY_DECLARATIONS
 int stonesoup_oc_i = 0;
 char stonesoup_function_buff[64];
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_start, "CWE590", "A", "Free of Memory not on
the Heap");
#endif /* STONESOUP_TRACE */
 strncpy(stonesoup_function_buff,STONESOUP_TAINT_SOURCE,63);
    stonesoup_function_buff[63] = '\0';
    for (; stonesoup_oc_i < 64; ++stonesoup_oc_i) {
        stonesoup_function_buff[stonesoup_oc_i] =
stonesoup_toupper(stonesoup_function_buff[stonesoup_oc_i]);
    }
    stonesoup_printf("%s\n",stonesoup_function_buff);
```

```
    stonesoup_free_func(stonesoup_function_buff);
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 ******************************************************************************/
int main(int argc,char *argv[])
{
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    char* tainted_buff = argv[1];

    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
    return 0;
```

```
}
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-590B Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1
#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1
#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1
#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1
#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1
#endif /* NOTHING DEFINED */

#if STONESOUP_SNIPPET_INCLUDES
/*****************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 *****************************************************************************/
#include <stdlib.h> /* malloc */
#include <stdio.h> /* fprintf */
#include <string.h> /* memset */
#include <errno.h> /* errno */
/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_INCLUDES */
#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 *****************************************************************************/
#include "output-handlers/handler.h"
```

```
#endif /* STONESOUP_SNIPPET_DEBUG */
char stonesoup_function_buff[64];
int stonesoup_toupper(int c) {
  if (c >= 97 && c <= 122) {
    return c - 32;
  }
  return c;
}
void stonesoup_free_func(char *buff) {
#if STONESOUP_TRACE
  tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE,
"stonesoup_free_func");
#endif /* STONESOUP_TRACE */
  if (buff[0] >= 79) {
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
    tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
    /* STONESOUP: CROSSOVER-POINT (Free of Invalid Pointer Not on the Heap) */
    /* STONESOUP: TRIGGER-POINT (Free of Invalid Pointer Not on the Heap) */
    free(buff);
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: AFTER");
    tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
  }
}

#endif /* STONESOUP_SNIPPET_SUPPORT */

#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 *****************************************************************************/
void weakness(char * stonesoup_tainted_buff) {
#define STONESOUP_TAINT_SOURCE stonesoup_tainted_buff
#endif /* STONESOUP_SNIPPET_DEBUG */
#if STONESOUP_SNIPPET_BODY_DECLARATIONS
 int stonesoup_oc_i = 0;
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_start, "CWE590", "B", "Free of Memory not on
the Heap");
#endif /* STONESOUP_TRACE */
    strncpy(stonesoup_function_buff,STONESOUP_TAINT_SOURCE,63);
    stonesoup_function_buff[63] = '\0';
    for (; stonesoup_oc_i < 64; ++stonesoup_oc_i) {
        stonesoup_function_buff[stonesoup_oc_i] =
stonesoup_toupper(stonesoup_function_buff[stonesoup_oc_i]);
    }
    stonesoup_printf("%s\n",stonesoup_function_buff);
```

```
    stonesoup_free_func(stonesoup_function_buff);
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 *******************************************************************************/
int main(int argc,char *argv[])
{
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    char* tainted_buff = argv[1];

    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
    return 0;
```

```
}
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-609A Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1

#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1

#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1

#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1

#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1

#endif /* NOTHING DEFINED */


#if STONESOUP_SNIPPET_INCLUDES
/******************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 ******************************************************************************/
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <string.h>

/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */

#endif /* STONESOUP_SNIPPET_INCLUDES */

#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
/******************************************************************************
```

```
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 ******************************************************************************/
#include "output-handlers/handler.h"

#endif /* STONESOUP_SNIPPET_DEBUG */

struct stonesoup_data {
    int qsize;
    int data_size;
    char *data;
    char *file1;
    char *file2;
};

struct stonesoup_data2 {
    int qsize;
    int data_size;
    int data_size2;
    char *data;
    char *data2;
};

pthread_mutex_t stonesoup_mutex;

int stonesoup_comp (const void * a, const void * b)
{
    if (a > b) {
        return -1;
    }
    else if (a < b) {
        return 1;
    }
    else {
        return 0;
    }
}

int stonesoup_pmoc (const void * a, const void * b)
{
    return -1 * stonesoup_comp(a, b);
}

void stonesoup_readFile(char *filename) {
    FILE *fifo;
    char ch;

#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE,
"stonesoup_readFile");
#endif /* STONESOUP_TRACE */

    fifo = fopen(filename, "r");
```

```
    if (fifo != NULL) {
        while ((ch = fgetc(fifo)) != EOF) {
            stonesoup_printf("%c", ch);
        }

        fclose(fifo);
    }
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "Finished reading sync file.");
#endif /* STONESOUP_TRACE */
}

struct stonesoup_data2 *ssD2 = 0;

struct stonesoup_data2 *init_stonesoup_data2 (struct stonesoup_data *ssD) {
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE,
"init_stonesoup_data2");
    tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */

    stonesoup_printf("Checking for initalization\n");
    /* STONESOUP: CROSSOVER-POINT (double-check locking) */
    if (ssD2 == NULL) {
        pthread_mutex_lock(&stonesoup_mutex);
        if (ssD2 == NULL) {
            stonesoup_printf("Initializing\n");

            ssD2 = calloc(1, sizeof(struct stonesoup_data2));

            stonesoup_readFile(ssD->file2);

            ssD2->data       = ssD->data;
            ssD2->qsize      = ssD->qsize;
            ssD2->data_size  = ssD->data_size;
            ssD2->data2      = ssD->data;
            ssD2->data_size2 = ssD->data_size;

        } else {
            stonesoup_printf("No need to initialize\n");
        }
        pthread_mutex_unlock(&stonesoup_mutex);
    } else {
        stonesoup_printf("Data is already initialized\n");
    }
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
#endif /* STONESOUP_TRACE */

    return ssD2;
}

void *doStuff(void *ssD) {
    struct stonesoup_data2 *ssD2;

#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE, "doStuff");
#endif /* STONESOUP_TRACE */
```

```
    stonesoup_printf("Inside doStuff\n");

    ssD2 = init_stonesoup_data2((struct stonesoup_data*)ssD);

    return NULL;
}

void *doStuff2(void *stonesoupData) {
    struct stonesoup_data2 *ssD2;
    struct stonesoup_data *ssD = stonesoupData;
    int stonesoup_i;
    int *stonesoup_arr;

#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE, "doStuff2");
#endif /* STONESOUP_TRACE */

    stonesoup_printf("Inside doStuff2\n");

    stonesoup_arr = malloc(sizeof(int) * ssD->qsize);
    for (stonesoup_i = 0; stonesoup_i < ssD->qsize; stonesoup_i++) {
        stonesoup_arr[stonesoup_i] = ssD->qsize - stonesoup_i;
    }
    qsort(stonesoup_arr, ssD->qsize, sizeof(int), &stonesoup_comp);
    free(stonesoup_arr);

    stonesoup_readFile(ssD->file1);

    ssD2 = init_stonesoup_data2((struct stonesoup_data*)ssD);

#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
    tracepoint(stonesoup_trace, variable_signed_integral, "ssD2->qsize", ssD2->qsize,
&(ssD2->qsize), "ssD2 - TRIGGER-STATE");
    tracepoint(stonesoup_trace, variable_buffer, "ssD2->data", ssD2->data, "ssD2 -
TRIGGER-STATE");
#endif /* STONESOUP_TRACE */

    /* STONESOUP: TRIGGER-POINT (double-check locking) */
    if (ssD2->data2[0] != '\0') {
        stonesoup_printf("%s\n", ssD2->data2);
    }

#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: AFTER");
#endif /* STONESOUP_TRACE */

    return NULL;
}
#endif /* STONESOUP_SNIPPET_SUPPORT */

#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
```

```
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 *****************************************************************************/
void weakness(char * taintedValue) {
#define STONESOUP_TAINT_SOURCE taintedValue
#endif /* STONESOUP_SNIPPET_DEBUG */
#if STONESOUP_SNIPPET_BODY_DECLARATIONS
    pthread_t stonesoup_t0, stonesoup_t1;
    struct stonesoup_data *stonesoupData;
    #endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_start, "CWE609", "A", "Double-Checked
Locking");
#endif /* STONESOUP_TRACE */

    stonesoupData = malloc(sizeof(struct stonesoup_data));
    if (stonesoupData) {
        stonesoupData->data  = malloc(sizeof(char) * (strlen(STONESOUP_TAINT_SOURCE) +
1));
        stonesoupData->file1 = malloc(sizeof(char) * (strlen(STONESOUP_TAINT_SOURCE) +
1));
        stonesoupData->file2 = malloc(sizeof(char) * (strlen(STONESOUP_TAINT_SOURCE) +
1));
        if (stonesoupData->data) {
            if ((sscanf(STONESOUP_TAINT_SOURCE, "%d %s %s %s",
                      &(stonesoupData->qsize),
                        stonesoupData->file1,
                        stonesoupData->file2,
                        stonesoupData->data)  == 4) &&
                (strlen(stonesoupData->data)  != 0) &&
                (strlen(stonesoupData->file1) != 0) &&
                (strlen(stonesoupData->file2) != 0))
            {
#if STONESOUP_TRACE
                tracepoint(stonesoup_trace, variable_signed_integral,
"stonesoupData->qsize", stonesoupData->qsize, &(stonesoupData->qsize), "stonesoupData
- INITIAL-STATE");
                tracepoint(stonesoup_trace, variable_buffer, "stonesoupData->data",
stonesoupData->data, "stonesoupData - INITIAL-STATE");
                tracepoint(stonesoup_trace, variable_buffer, "stonesoupData->file1",
stonesoupData->file1, "stonesoupData - INITIAL-STATE");
                tracepoint(stonesoup_trace, variable_buffer, "stonesoupData->file2",
stonesoupData->file2, "stonesoupData - INITIAL-STATE");
#endif /* STONESOUP_TRACE */

                stonesoupData->data_size = strlen(stonesoupData->data);
                if (pthread_mutex_init(&stonesoup_mutex, NULL) != 0) {
                    stonesoup_printf("Mutex failed to initilize.");
                 }

#if STONESOUP_TRACE
                tracepoint(stonesoup_trace, trace_point, "Spawning threads.");
#endif /* STONESOUP_TRACE */

                if (pthread_create(&stonesoup_t0, NULL, doStuff, (void
*)stonesoupData) != 0) {
                    stonesoup_printf("Error creating thread 0.");
                }
```

```
                if (pthread_create(&stonesoup_t1, NULL, doStuff2, (void
*)stonesoupData) != 0) {
                    stonesoup_printf("Error creating thread 1.");
                }

                pthread_join(stonesoup_t0, NULL);
                pthread_join(stonesoup_t1, NULL);

#if STONESOUP_TRACE
                tracepoint(stonesoup_trace, trace_point, "Threads joined.");
#endif /* STONESOUP_TRACE */

                pthread_mutex_destroy(&stonesoup_mutex);
            }
            free(stonesoupData->data);
        }
        free(stonesoupData);
    } else {
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, trace_error, "Error parsing input.");
#endif /* STONESOUP_TRACE */
        stonesoup_printf("Error parsing input.\n");
    }
    if (ssD2) {
        free (ssD2);
    }
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */

#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 ******************************************************************************/
int main(int argc,char *argv[])
{
    char* tainted_buff;
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    tainted_buff = argv[1];

    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
```

```
    return 0;
}
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-663A Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1

#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1

#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1

#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1

#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1

#endif /* NOTHING DEFINED */


#if STONESOUP_SNIPPET_INCLUDES
/*****************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 *****************************************************************************/
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <string.h>

/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */

#endif /* STONESOUP_SNIPPET_INCLUDES */

#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
```

```
/*******************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 ******************************************************************************/
#include "output-handlers/handler.h"

#endif /* STONESOUP_SNIPPET_DEBUG */

struct stonesoup_data {
    int qsize;
    int data_size;
    char *data;
};

pthread_mutex_t stonesoup_mutex;

int stonesoup_comp (const void * a, const void * b) {
    if (a > b) {
        return -1;
    }
    else if (a < b) {
        return 1;
    }
    else {
        return 0;
    }
}

int stonesoup_pmoc (const void * a, const void * b) {
    return -1 * stonesoup_comp(a, b);
}

void arrFunc (struct stonesoup_data *stonesoupData) {
    int *stonesoup_arr = malloc(sizeof(int) * stonesoupData->qsize);
    int stonesoup_i;
    int stonesoup_j;
    FILE *fp;
    static int stonesoup_ctr;

#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE, "arrFunc");
    tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
    stonesoup_ctr = 0;                                                      /* static
var will reset to 0 */
    for(stonesoup_i = 0;                                                    /* and
count twice when second thread is in func */
        stonesoup_i < stonesoupData->data_size;                             /* which
will blow the free() struct away */
        stonesoup_i++, stonesoup_ctr++)
    {
        /* STONESOUP: CROSSOVER-POINT (nonreentrant function in multi-threaded
context) */
        stonesoupData->data[stonesoup_ctr] = '\0';
```

```
        stonesoup_printf("I: %d, C: %d\n", stonesoup_i, stonesoup_ctr);

        if (stonesoupData->qsize > 5) {
            fp = fopen("asdfqwer1234", "w+");
            fprintf(fp, "%d", stonesoup_i);
            fclose(fp);

            for (stonesoup_j = 0; stonesoup_j < stonesoupData->qsize; stonesoup_j++) {
                stonesoup_arr[stonesoup_j] = stonesoupData->qsize - stonesoup_j;
            }
            qsort(stonesoup_arr, stonesoupData->qsize, sizeof(int), &stonesoup_comp);
        }
    }
    free(stonesoup_arr);
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, variable_signed_integral, "stonesoupData->data_size",
stonesoupData->data_size, &stonesoupData->data_size, "CROSSOVER-STATE");
    tracepoint(stonesoup_trace, variable_signed_integral, "stonesoup_ctr",
stonesoup_ctr, &stonesoup_ctr, "CROSSOVER-STATE");
    tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
}

void *replaceSymbols(void *data) {
    struct stonesoup_data *stonesoupData = data;
    int i;

#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE,
"replaceSymbols");
#endif /* STONESOUP_TRACE */

    pthread_mutex_lock(&stonesoup_mutex);
    for(i = 0; i < stonesoupData->data_size; i++) {
        if (((stonesoupData->data[i] >= '!' && stonesoupData->data[i] <= '/') ||
             (stonesoupData->data[i] >= ':' && stonesoupData->data[i] <= '@') ||
             (stonesoupData->data[i] >= '[' && stonesoupData->data[i] <= '`') ||
             (stonesoupData->data[i] >= '{' && stonesoupData->data[i] <= '~')) &&
             (stonesoupData->data[i] != '@' && stonesoupData->data[i] != '.'))
        {
            stonesoupData->data[i] = '_';
        }
    }
    pthread_mutex_unlock(&stonesoup_mutex);
    arrFunc(stonesoupData);

    return NULL;
}

void *toCaps(void *data) {
    struct stonesoup_data *stonesoupData = data;
    int threadTiming = 500000;
    int stonesoup_j;
    int *stonesoup_arr;
    int i;

#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE, "toCaps");
```

Approved for public release; distribution unlimited.

```
#endif /* STONESOUP_TRACE */

    /* slow things down to make correct thing happen in good cases */
    stonesoup_arr = malloc(sizeof(int)*threadTiming);
    if (stonesoup_arr != NULL) {
        for (stonesoup_j = 0; stonesoup_j < threadTiming; stonesoup_j++) {
            stonesoup_arr[stonesoup_j] = threadTiming - stonesoup_j;
        }
        qsort(stonesoup_arr, threadTiming, sizeof(int), &stonesoup_comp);
        free (stonesoup_arr);
        stonesoup_arr = NULL;
    }

    pthread_mutex_lock(&stonesoup_mutex);

    for(i = 0; i < stonesoupData->data_size; i++) {
        if(stonesoupData->data[i] >= 'a' && stonesoupData->data[i] <= 'z') {
            stonesoupData->data[i] -= 'a' - 'A';
        }
    }
    pthread_mutex_unlock(&stonesoup_mutex);
    arrFunc(stonesoupData);

    return NULL;
}

#endif /* STONESOUP_SNIPPET_SUPPORT */

#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/******************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 ******************************************************************************/
void weakness(char * stonesoup_tainted_buff) {
#define STONESOUP_TAINT_SOURCE stonesoup_tainted_buff
#endif /* STONESOUP_SNIPPET_DEBUG */

#if STONESOUP_SNIPPET_BODY_DECLARATIONS
    pthread_t stonesoup_t0, stonesoup_t1;
    struct stonesoup_data *stonesoupData = malloc(sizeof(struct stonesoup_data));
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_start, "CWE663", "A", "Use of a Non-reentrant
Function in a Concurrent Context");
#endif /* STONESOUP_TRACE */
    if (stonesoupData) {
        stonesoupData->data = malloc(sizeof(char) * (strlen(STONESOUP_TAINT_SOURCE)+
1));
        if (stonesoupData->data &&
            (sscanf(STONESOUP_TAINT_SOURCE, "%d %s", &stonesoupData->qsize,
stonesoupData->data) == 2) &&
            (strlen(stonesoupData->data) != 0)) {
```

```
            pthread_mutex_init(&stonesoup_mutex, NULL);

            stonesoupData->data_size = strlen(stonesoupData->data);

#if STONESOUP_TRACE
            tracepoint(stonesoup_trace, variable_signed_integral,
"stonesoupData->qsize", stonesoupData->qsize, &stonesoupData->qsize, "INITIAL-STATE");
            tracepoint(stonesoup_trace, variable_buffer, "stonesoupData->data",
stonesoupData->data, "INTIAL-STATE");
            tracepoint(stonesoup_trace, variable_signed_integral,
"stonesoupData->data_size", stonesoupData->data_size, &stonesoupData->data_size,
"INTIAL-STATE");
#endif /* STONESOUP_TRACE */

#if STONESOUP_TRACE
            tracepoint(stonesoup_trace, trace_point, "Spawning threads");
#endif /* STONESOUP_TRACE */

            if (pthread_create(&stonesoup_t0, NULL, replaceSymbols,
(void*)stonesoupData) != 0) {
                stonesoup_printf("Error initilizing thread 0.");
            }

            if (pthread_create(&stonesoup_t1, NULL, toCaps, (void*)stonesoupData) !=
0) {
                stonesoup_printf("Error initilizing thread 1.");
            }

            pthread_join(stonesoup_t0, NULL);
            pthread_join(stonesoup_t1, NULL);

#if STONESOUP_TRACE
            tracepoint(stonesoup_trace, trace_point, "Threads joined.");
#endif /* STONESOUP_TRACE */

            pthread_mutex_destroy(&stonesoup_mutex);

#if STONESOUP_TRACE
            tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
            /* STONESOUP: TRIGGER-POINT (nonreentrant function in multi-threaded
context) */
            free(stonesoupData->data);
#if STONESOUP_TRACE
            tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
        }
        else {
            stonesoup_printf("Error parsing input.\n");
        }
        free(stonesoupData);
    }
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
```

```
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */

#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 ******************************************************************************/
int main(int argc,char *argv[])
{
    char* tainted_buff;
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    tainted_buff = argv[1];

    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
```

```
    return 0;
}
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-682A Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1
#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1
#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1
#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1
#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1
#endif /* NOTHING DEFINED */

#if STONESOUP_SNIPPET_INCLUDES
/*******************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 ******************************************************************************/
#include <stdlib.h> // malloc
#include <stdio.h> // vfprintf
#include <string.h> // strlen
/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_INCLUDES */
#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 ******************************************************************************/
#include "output-handlers/handler.h"
```

```
#endif /* STONESOUP_SNIPPET_DEBUG */
void stonesoup_function() {
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE,
"stonesoup_function");
#endif /* STONESOUP_TRACE */
}
#endif /* STONESOUP_SNIPPET_SUPPORT */
#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/******************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 ******************************************************************************/
void weakness(char * stonesoup_tainted_buff) {
#define STONESOUP_TAINT_SOURCE stonesoup_tainted_buff
#endif /* STONESOUP_SNIPPET_DEBUG */
#if STONESOUP_SNIPPET_BODY_DECLARATIONS
    void (*stonesoup_function_ptr_1)() = 0;
    void (*stonesoup_function_ptr_2)() = 0;
    unsigned long stonesoup_input_num;
    void (*stonesoup_function_ptr_3)() = 0;
    void (*stonesoup_function_ptr_4)() = 0;
    char *stonesoup_byte_4 = 0;
    char *stonesoup_byte_3 = 0;
    unsigned long *stonesoup_ptr = 0;
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_start, "CWE682", "A", "Incorrect
Calculation");
#endif /* STONESOUP_TRACE */
    stonesoup_function_ptr_1 = stonesoup_function;
    stonesoup_function_ptr_2 = stonesoup_function;
    stonesoup_function_ptr_3 = stonesoup_function;
    stonesoup_function_ptr_4 = stonesoup_function;

    if (strlen(STONESOUP_TAINT_SOURCE) >= 1 &&
            STONESOUP_TAINT_SOURCE[0] != '-') {
        stonesoup_input_num = strtoul(STONESOUP_TAINT_SOURCE,0U,16);
        stonesoup_ptr = &stonesoup_input_num;

        if ( *stonesoup_ptr > 65535) {
#if STONESOUP_TRACE
            tracepoint(stonesoup_trace, variable_address, "&stonesoup_function_ptr_1",
&stonesoup_function_ptr_1, "INITIAL-STATE");
            tracepoint(stonesoup_trace, variable_address, "&stonesoup_function_ptr_2",
&stonesoup_function_ptr_2, "INITIAL-STATE");
            tracepoint(stonesoup_trace, variable_address, "&stonesoup_input_num",
&stonesoup_input_num, "INITIAL-STATE");
            tracepoint(stonesoup_trace, variable_address, "&stonesoup_function_ptr_3",
&stonesoup_function_ptr_3, "INITIAL-STATE");
            tracepoint(stonesoup_trace, variable_address, "&stonesoup_function_ptr_4",
&stonesoup_function_ptr_4, "INITIAL-STATE");
```

```
            tracepoint(stonesoup_trace, variable_address, "&stonesoup_byte_4",
&stonesoup_byte_4, "INITIAL-STATE");
            tracepoint(stonesoup_trace, variable_address, "&stonesoup_byte_3",
&stonesoup_byte_3, "INITIAL-STATE");
            tracepoint(stonesoup_trace, variable_address, "&stonesoup_ptr",
&stonesoup_ptr, "INITIAL-STATE");
            tracepoint(stonesoup_trace, variable_address, "stonesoup_function_ptr_1",
stonesoup_function_ptr_1, "INITIAL-STATE");
            tracepoint(stonesoup_trace, variable_address, "stonesoup_function_ptr_2",
stonesoup_function_ptr_2, "INITIAL-STATE");
            tracepoint(stonesoup_trace, variable_unsigned_integral,
"&stonesoup_input_num", stonesoup_input_num, &stonesoup_input_num, "INITIAL-STATE");
            tracepoint(stonesoup_trace, variable_address, "stonesoup_function_ptr_3",
stonesoup_function_ptr_3, "INITIAL-STATE");
            tracepoint(stonesoup_trace, variable_address, "stonesoup_function_ptr_4",
stonesoup_function_ptr_4, "INITIAL-STATE");
            tracepoint(stonesoup_trace, variable_address, "stonesoup_byte_4",
stonesoup_byte_4, "INITIAL-STATE");
            tracepoint(stonesoup_trace, variable_address, "stonesoup_byte_3",
stonesoup_byte_3, "INITIAL-STATE");
            tracepoint(stonesoup_trace, variable_unsigned_integral, "*stonesoup_ptr",
*stonesoup_ptr, stonesoup_ptr, "INITIAL-STATE");
            tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
            /* STONESOUP: CROSSOVER-POINT (Incorrect Calculation) */
            stonesoup_byte_3 = ((char *)(stonesoup_ptr + 2));
            stonesoup_byte_4 = ((char *)(stonesoup_ptr + 3));
#if STONESOUP_TRACE
            tracepoint(stonesoup_trace, variable_address, "stonesoup_byte_3",
stonesoup_byte_3, "CROSSOVER-STATE");
            tracepoint(stonesoup_trace, variable_address, "stonesoup_byte_4",
stonesoup_byte_4, "CROSSOVER-STATE");
#endif /* STONESOUP_TRACE */
             *stonesoup_byte_3 = 0;
             *stonesoup_byte_4 = 0;
#if STONESOUP_TRACE
            tracepoint(stonesoup_trace, variable_address, "stonesoup_function_ptr_1",
stonesoup_function_ptr_1, "CROSSOVER-STATE");
            tracepoint(stonesoup_trace, variable_address, "stonesoup_function_ptr_2",
stonesoup_function_ptr_2, "CROSSOVER-STATE");
            tracepoint(stonesoup_trace, variable_address, "stonesoup_function_ptr_3",
stonesoup_function_ptr_3, "CROSSOVER-STATE");
            tracepoint(stonesoup_trace, variable_address, "stonesoup_function_ptr_4",
stonesoup_function_ptr_4, "CROSSOVER-STATE");
            tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
        }
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
        /* STONESOUP: TRIGGER-POINT (Incorrect Calculation) */
        stonesoup_function_ptr_1();
        stonesoup_function_ptr_2();
        stonesoup_function_ptr_3();
        stonesoup_function_ptr_4();
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
        stonesoup_printf("Value = %i\n", stonesoup_input_num);
```

```
    } else if (strlen(STONESOUP_TAINT_SOURCE) == 0) {
        stonesoup_printf("Input is empty string\n");
    } else {
        stonesoup_printf("Input is negative number\n");
    }
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_DEBUG
/******************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 ******************************************************************************/
int main(int argc,char *argv[])
{
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    char* tainted_buff = argv[1];

    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
    exit(0);
```

```
}
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-682B Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1
#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1
#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1
#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1
#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1
#endif /* NOTHING DEFINED */

#if STONESOUP_SNIPPET_INCLUDES
/*****************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 *****************************************************************************/
#include <stdlib.h> // malloc
#include <stdio.h> // vfprintf
#include <string.h> // strlen
/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_INCLUDES */
#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 *****************************************************************************/
/*
```

```
 * New algorithmic variant to deal with stack reordering in GCC 4.6.
 * GCC 4.6 reorders variables on the stack in a way to breaks the
 * previous variant, using a struct allocated on the stack we are
 * able to guarantee a contiguous memory block in this version,
 * preserving the intended functionality on GCC 4.6.
 */
#include "output-handlers/handler.h"
#endif /* STONESOUP_SNIPPET_DEBUG */
struct stonesoup_struct {
    void (*stonesoup_function_ptr_1)();
    unsigned int stonesoup_input_num;
    void (*stonesoup_function_ptr_2)();
};
void stonesoup_function() {
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE,
"stonesoup_function");
#endif /* STONESOUP_TRACE */
}
#endif /* STONESOUP_SNIPPET_SUPPORT */
#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/******************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 ******************************************************************************/
void weakness(char * stonesoup_tainted_buff) {
#define STONESOUP_TAINT_SOURCE stonesoup_tainted_buff
#endif /* STONESOUP_SNIPPET_DEBUG */
#if STONESOUP_SNIPPET_BODY_DECLARATIONS
    char *stonesoup_byte_4 = 0;
    char *stonesoup_byte_3 = 0;
    unsigned int *stonesoup_ptr = 0;
    struct stonesoup_struct ssS;
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_start, "CWE682", "B", "Incorrect
Calculation");
#endif /* STONESOUP_TRACE */
    ssS.stonesoup_function_ptr_1 = stonesoup_function;
    ssS.stonesoup_function_ptr_2 = stonesoup_function;

    if (strlen(STONESOUP_TAINT_SOURCE) >= 1 &&
            STONESOUP_TAINT_SOURCE[0] != '-') {
        ssS.stonesoup_input_num = strtoul(STONESOUP_TAINT_SOURCE,0U,16);
        stonesoup_ptr = &(ssS.stonesoup_input_num);

        if ( *stonesoup_ptr > 65535) {
#if STONESOUP_TRACE
            tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
            tracepoint(stonesoup_trace, variable_address,
"(ssS.stonesoup_function_ptr_2)", (ssS.stonesoup_function_ptr_2), "INITIAL-STATE");
#endif /* STONESOUP_TRACE */
```

```
            /* STONESOUP: CROSSOVER-POINT (Incorrect Calculation) */
            stonesoup_byte_3 = ((char *)(stonesoup_ptr + 2));
            stonesoup_byte_4 = ((char *)(stonesoup_ptr + 3));
             *stonesoup_byte_3 = 0;
             *stonesoup_byte_4 = 0;
#if STONESOUP_TRACE
            tracepoint(stonesoup_trace, variable_address,
"(ssS.stonesoup_function_ptr_2)", (ssS.stonesoup_function_ptr_2), "CROSSOVER-STATE");
#endif /* STONESOUP_TRACE */
            /* STONESOUP: CROSSOVER-POINT (Incorrect Calculation) */
#if STONESOUP_TRACE
            tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
        }
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
        /* STONESOUP: TRIGGER-POINT (Incorrect Calculation) */
        ssS.stonesoup_function_ptr_2();
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
        stonesoup_printf("Value = %i\n", ssS.stonesoup_input_num);
    } else if (strlen(STONESOUP_TAINT_SOURCE) == 0) {
        stonesoup_printf("Input is empty string\n");
    } else {
        stonesoup_printf("Input is negative number\n");
    }
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_DEBUG
/******************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 ******************************************************************************/
int main(int argc,char *argv[])
{
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    char* tainted_buff = argv[1];

    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
    exit(0);
```

```
}
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-761A Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1
#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1
#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1
#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1
#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1
#endif /* NOTHING DEFINED */

#if STONESOUP_SNIPPET_INCLUDES
/*******************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 ******************************************************************************/
#include <stdlib.h> /* malloc */
#include <stdio.h> /* fprintf */
#include <string.h> /* memset */
#include <errno.h> /* errno */
/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_INCLUDES */
#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 ******************************************************************************/
```

```
#include "output-handlers/handler.h"
#endif /* STONESOUP_SNIPPET_DEBUG */
int stonesoup_contains_char(char *str_param,char c_param)
{
#if STONESOUP_TRACE
  tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE,
"stonesoup_contains_char");
#endif /* STONESOUP_TRACE */
  int function_found;
  function_found = 0;
#if STONESOUP_TRACE
  tracepoint(stonesoup_trace, variable_address, "str_param", str_param,
"INITIAL-STATE");
  tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
/* STONESOUP: CROSSOVER-POINT (Free Not At Start Of Buffer) */
  while( *str_param != 0){
    if ( *str_param == c_param) {
      function_found = 1;
      break;
    }
    str_param = str_param + 1;
  }
#if STONESOUP_TRACE
  tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
  tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
  tracepoint(stonesoup_trace, variable_address, "str_param", str_param,
"TRIGGER-STATE");
#endif /* STONESOUP_TRACE */
/* STONESOUP: TRIGGER-POINT (Free Not At Start Of Buffer) */
  free(str_param);
#if STONESOUP_TRACE
  tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
  return function_found;
}
int stonesoup_toupper(int c)
{
  if (c >= 97 && c <= 122) {
    return c - 32;
  }
  return c;
}
#endif /* STONESOUP_SNIPPET_SUPPORT */

#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 ******************************************************************************/
void weakness(char * stonesoup_tainted_buff) {
#define STONESOUP_TAINT_SOURCE stonesoup_tainted_buff
#endif /* STONESOUP_SNIPPET_DEBUG */
```

```
#if STONESOUP_SNIPPET_BODY_DECLARATIONS
 int stonesoup_oc_i = 0;
 int stonesoup_found;
 char *stonesoup_buffer = 0;
 int stonesoup_buffer_len;
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_start, "CWE761", "A", "Free of Pointer not at
Start of Buffer");
#endif /* STONESOUP_TRACE */
    stonesoup_buffer_len = strlen(STONESOUP_TAINT_SOURCE) + 1;
    stonesoup_buffer = malloc(stonesoup_buffer_len * sizeof(char ));
    if (stonesoup_buffer == 0) {
        stonesoup_printf("Error: Failed to allocate memory\n");
        exit(1);
    }
    strcpy(stonesoup_buffer,STONESOUP_TAINT_SOURCE);
    for (; stonesoup_oc_i < stonesoup_buffer_len; ++stonesoup_oc_i) {
        stonesoup_buffer[stonesoup_oc_i] =
stonesoup_toupper(stonesoup_buffer[stonesoup_oc_i]);
    }
    stonesoup_printf("%s\n",stonesoup_buffer);
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, variable_buffer, "stonesoup_buffer", stonesoup_buffer,
"INITIAL_STATE");
#endif /* STONESOUP_TRACE */
    stonesoup_found = stonesoup_contains_char(stonesoup_buffer,'E');
    if (stonesoup_found == 1)
        stonesoup_printf("%s\n",STONESOUP_TAINT_SOURCE);
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 ******************************************************************************/
int main(int argc,char *argv[])
{
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    char* tainted_buff = argv[1];

    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
    return 0;
```

Approved for public release; distribution unlimited.

```
}
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-764A Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1

#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1

#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1

#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1

#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1

#endif /* NOTHING DEFINED */


#if STONESOUP_SNIPPET_INCLUDES
/*****************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 *****************************************************************************/
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <string.h>
#include <semaphore.h>

/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */

#endif /* STONESOUP_SNIPPET_INCLUDES */

#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
```

```
/*******************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 ******************************************************************************/
#include "output-handlers/handler.h"

#endif /* STONESOUP_SNIPPET_DEBUG */

sem_t stonesoup_sem;
pthread_t stonesoup_t0, stonesoup_t1;
char *stonesoup_global_str;

int stonesoup_isspace(char c) {
    return (c == ' ' || c == '\t' || c == '\n');
}

void *replaceSpace () {
    int stonesoup_i = 0;
    stonesoup_printf("Replacing spaces\n");

#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE,
"replaceSpace");
    tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
    /* STONESOUP: TRIGGER-POINT (multiple locks) */
    sem_wait(&stonesoup_sem);                                   /* multiple locks -
deadlock */
    sem_wait(&stonesoup_sem);
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: AFTER");
#endif /* STONESOUP_TRACE */

    while(stonesoup_global_str[stonesoup_i] != '\0') {
        if (stonesoup_isspace(stonesoup_global_str[stonesoup_i]) != 0) {
            stonesoup_global_str[stonesoup_i] = '_';
        }
        stonesoup_i++;
    }

    sem_post(&stonesoup_sem);
    return NULL;
}

void *toCap () {
    int stonesoup_i = 0;

#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE, "toCap");
    tracepoint(stonesoup_trace, trace_point, "Before sem_wait in toCap");
#endif /* STONESOUP_TRACE */
    stonesoup_printf("Capitalizing input\n");
    sem_wait(&stonesoup_sem);
#if STONESOUP_TRACE
```

```
    tracepoint(stonesoup_trace, trace_point, "After sem_wait in toCap");
#endif /* STONESOUP_TRACE */

    while(stonesoup_global_str[stonesoup_i] != '\0') {
        if (stonesoup_global_str[stonesoup_i] > 'a' &&
stonesoup_global_str[stonesoup_i] < 'z') {
            stonesoup_global_str[stonesoup_i] -= 'a' - 'A';
        }
        stonesoup_i++;
    }

    sem_post(&stonesoup_sem);
    return NULL;
}

#endif /* STONESOUP_SNIPPET_SUPPORT */

#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 *****************************************************************************/
void weakness(char * stonesoup_tainted_buff) {
#define STONESOUP_TAINT_SOURCE stonesoup_tainted_buff
#endif /* STONESOUP_SNIPPET_DEBUG */

#if STONESOUP_SNIPPET_BODY_DECLARATIONS
    int stonesoup_hasSpaces = 0;
    int stonesoup_i = 0;
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_start, "CWE764", "A", "Multiple Locks of a
Critical Resource");
#endif /* STONESOUP_TRACE */
    sem_init(&stonesoup_sem, 0, 1);

    while(STONESOUP_TAINT_SOURCE[stonesoup_i] != '\0') {                          /*
if the input contains spaces */
        if (stonesoup_isspace(STONESOUP_TAINT_SOURCE[stonesoup_i++]) != 0) {      /*
we will call the deadlocking function */
            stonesoup_hasSpaces = 1;
        }
    }

#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, variable_buffer, "STONESOUP_TAINT_SOURCE",
STONESOUP_TAINT_SOURCE, "INITIAL-STATE");
#endif /* STONESOUP_TRACE */

    stonesoup_global_str = malloc(sizeof(char) * strlen(STONESOUP_TAINT_SOURCE) + 1);
    strcpy(stonesoup_global_str, STONESOUP_TAINT_SOURCE);
```

```
    if (stonesoup_hasSpaces == 1) {
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
        /* STONESOUP: CROSSOVER-POINT (multiple locks) */
        if (pthread_create(&stonesoup_t0, NULL, replaceSpace, NULL) != 0) {
            stonesoup_printf("Thread 0 failed to spawn.");
        }
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
    }
    if (pthread_create(&stonesoup_t1, NULL, toCap, NULL) != 0) {
        stonesoup_printf("Thread 1 failed to spawn.");
    }

    if (stonesoup_hasSpaces == 1) {
        pthread_join(stonesoup_t0, NULL);
    }
    pthread_join(stonesoup_t1, NULL);
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */

#if STONESOUP_SNIPPET_DEBUG
/******************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 *****************************************************************************/
int main(int argc,char *argv[])
{
    char* tainted_buff;
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    tainted_buff = argv[1];

    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
```

```
    return 0;
}
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-765A Source Code

```
 #if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1

#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1

#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1

#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1

#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1

#endif /* NOTHING DEFINED */


#if STONESOUP_SNIPPET_INCLUDES
/******************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 ******************************************************************************/
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <stdlib.h>
#include <string.h>

#endif /* STONESOUP_SNIPPET_INCLUDES */

#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
/******************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 ******************************************************************************/
#include "output-handlers/handler.h"
```

```
#endif /* STONESOUP_SNIPPET_DEBUG */

struct stonesoup_data {
    int qsize;
    char *file1;
    char *file2;
    char *data;
    int data_size;
};

pthread_t stonesoup_t0, stonesoup_t1;
sem_t stonesoup_sem;

int stonesoup_comp (const void * a, const void * b)
{
    if (a > b) {
        return -1;
    }
    else if (a < b) {
        return 1;
    }
    else {
        return 0;
    }
}

int stonesoup_pmoc (const void * a, const void * b)
{
    return -1 * stonesoup_comp(a, b);
}

void stonesoup_readFile(char *filename) {
    FILE *fifo;
    char ch;

    fifo = fopen(filename, "r");

    if (fifo != NULL) {
        while ((ch = fgetc(fifo)) != EOF) {
            stonesoup_printf("%c", ch);
        }

        fclose(fifo);
    }
}

void *to1337(void *data) {
    struct stonesoup_data *stonesoupData = (struct stonesoup_data*)data;
    int qsize;
    int random;
    char temp;
    char *temp_str;
    int i = 0;
    int *stonesoup_arr;

    stonesoup_printf("Entering to1337\n");

    /* slow things down to make correct thing happen in good cases */
```

```
    stonesoup_arr = malloc(sizeof(int)*stonesoupData->qsize);
    if (stonesoup_arr != NULL) {
        for (i = 0; i < stonesoupData->qsize; i++) {
            stonesoup_arr[i] = stonesoupData->qsize - i;
        }
        qsort(stonesoup_arr, stonesoupData->qsize, sizeof(int), &stonesoup_comp);
        free (stonesoup_arr);
        stonesoup_arr = NULL;
    }

    temp_str = malloc(sizeof(char)*(stonesoupData->data_size + 1));
    qsize = stonesoupData->qsize;

    sem_wait(&stonesoup_sem);

    i = 0;
    while(stonesoupData->data[i] != '\0') {
        random = (int)(rand() / (double)RAND_MAX + 0.5);    /* add .5 before
truncation to round */
        switch(stonesoupData->data[i]) {                    /* 1337 s<r1p7 i5 f0r
h4x0r5 */
            case 'c':
                if (random == 0)
                    temp = '<';
                else
                    temp = 'c';
                break;
            case 'e':
                if (random == 0)
                    temp = '3';
                else
                    temp = 'e';
                break;
            case 'i':
                if (random == 0)
                    temp = '1';
                else
                    temp = 'i';
                break;
            case 'l':
                if (random == 0)
                    temp = '1';
                else
                    temp = 'l';
                break;
            case 'o':
                if (random == 0)
                    temp = '0';
                else
                    temp = 'o';
                break;
            case 's':
                if (random == 0)
                    temp = '5';
                else
                    temp = 's';
                break;
            case 't':
                if (random == 0)
```

```
                    temp = '7';
                else
                    temp = 't';
                break;
            default:
                temp = stonesoupData->data[i];
                break;
        }
        temp_str[i] = temp;

        i++;
    }
    temp_str[i] = '\0';

    free(stonesoupData->data);
    stonesoupData->data = NULL;                                   /* setting free()'d
ptrs to null is good practice yo */

    stonesoup_printf("Set ptr to null\n");

    /* execute second */
    stonesoup_readFile(stonesoupData->file2);

    stonesoup_printf("Set ptr to NON null\n");
    stonesoupData->data = temp_str;
    sem_post(&stonesoup_sem);
    return NULL;
}

void *reverseStr(void * data) {
    struct stonesoup_data *stonesoupData = (struct stonesoup_data*)data;
    int i = 0;
    char *temp_str;

    stonesoup_printf("Entering reverseStr\n");

    /* execute first */
    stonesoup_readFile(stonesoupData->file1);

    sem_wait(&stonesoup_sem);                                      /* if weakness has
been triggered, */
                                                                   /* too many
resource copies available */
    temp_str = malloc(sizeof(char)* (stonesoupData->data_size + 1));

    for (i = 0; i < stonesoupData->data_size; i++) {
        /* STONESOUP: TRIGGER-POINT (multipleunlocks) */
        stonesoup_printf("Dereferencing ptr\n");
        temp_str[stonesoupData->data_size - 1 - i] = stonesoupData->data[i];    /*
null ptr dereference */
    }

    temp_str[stonesoupData->data_size] = '\0';

    free(stonesoupData->data);
    stonesoupData->data = NULL;
    stonesoupData->data = temp_str;
    sem_post(&stonesoup_sem);
    return NULL;
```

Approved for public release; distribution unlimited.

```
}

void toLower (struct stonesoup_data * stonesoupData) {
    int i = 0;
    sem_wait(&stonesoup_sem);
    stonesoup_printf("Entering toLower\n");

    for (i = 0; i < strlen(stonesoupData->data) - 1; i++) {  /* all caps to lower */
        if (stonesoupData->data[i] >= 'A' &&
            stonesoupData->data[i] <= 'Z') {
            stonesoupData->data[i] += 32;
        }
    }

    sem_post(&stonesoup_sem);
    /* STONESOUP: CROSSOVER-POINT (multipleunlocks) */
    sem_post(&stonesoup_sem);                                 /* oops, extra unlock */
}

#endif /* STONESOUP_SNIPPET_SUPPORT */

#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 *****************************************************************************/
void weakness(char * stonesoup_taintedValue) {
#define STONESOUP_TAINT_SOURCE stonesoup_taintedValue
#endif /* STONESOUP_SNIPPET_DEBUG */

#if STONESOUP_SNIPPET_BODY_DECLARATIONS
    int hasCap = 0;
    int stonesoup_i = 0;
    struct stonesoup_data *stonesoupData;
 #endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */

#if STONESOUP_SNIPPET_BODY_STATEMENTS
    stonesoupData =  malloc(sizeof(struct stonesoup_data));
    if (stonesoupData) {
        stonesoupData->data  = malloc(sizeof(char) * (strlen(STONESOUP_TAINT_SOURCE) +
1));
        stonesoupData->file1 = malloc(sizeof(char) * (strlen(STONESOUP_TAINT_SOURCE) +
1));
        stonesoupData->file2 = malloc(sizeof(char) * (strlen(STONESOUP_TAINT_SOURCE) +
1));
        if (stonesoupData->data) {
            if ((sscanf(STONESOUP_TAINT_SOURCE, "%d %s %s %s",
               &(stonesoupData->qsize),
                 stonesoupData->file1,
                 stonesoupData->file2,
                 stonesoupData->data) == 4) &&
                (strlen(stonesoupData->data) != 0))
            {
```

```
                sem_init(&stonesoup_sem, 0, 1);

                stonesoupData->data_size = strlen(stonesoupData->data);
                while (stonesoupData->data[stonesoup_i] != '\0') {       /* if input
has capital */
                    if (stonesoupData->data[stonesoup_i] >= 'A' &&       /*   call
function that contains */
                        stonesoupData->data[stonesoup_i] <= 'Z') {       /*   weakness
*/
                        hasCap = 1;
                    }
                    stonesoup_i++;
                }

                if (hasCap == 1) {
                    toLower(stonesoupData);
                }

                if (pthread_create(&stonesoup_t0, NULL, reverseStr, (void
*)stonesoupData) != 0) {
                    stonesoup_printf("Error creating thread 0.");
                }


                if (pthread_create(&stonesoup_t1, NULL, to1337, (void *)stonesoupData)
!= 0) {
                    stonesoup_printf("Error creating thread 1.");
                }

                pthread_join(stonesoup_t0, NULL);
                pthread_join(stonesoup_t1, NULL);

                stonesoup_printf("After joins.\n");
                stonesoup_printf("String: %s\n", stonesoupData->data);
            }
            free(stonesoupData->data);
        }
        free(stonesoupData);
    } else {
        stonesoup_printf("Error parsing input.\n");
    }
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */

#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 ******************************************************************************/
int main(int argc,char *argv[])
{
    char* tainted_buff;
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
```

```
    }
    tainted_buff = argv[1];

    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
```

Approved for public release; distribution unlimited.

```
    return 0;
}
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-765B Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1

#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1

#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1

#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1

#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1

#endif /* NOTHING DEFINED */


#if STONESOUP_SNIPPET_INCLUDES
/******************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 ******************************************************************************/
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <stdlib.h>
#include <string.h>

/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */

#endif /* STONESOUP_SNIPPET_INCLUDES */

#if STONESOUP_SNIPPET_SUPPORT
```

```
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 *****************************************************************************/
#include "output-handlers/handler.h"

#endif /* STONESOUP_SNIPPET_DEBUG */

pthread_t stonesoup_t0, stonesoup_t1, stonesoup_t2;
sem_t stonesoup_sem;
struct stonesoup_data {
    int qsize;
    char *data;
    char *file1;
    char *file2;
};


int stonesoup_comp (const void * a, const void * b)
{
    if (a > b) {
        return -1;
    }
    else if (a < b) {
        return 1;
    }
    else {
        return 0;
    }
}

int stonesoup_pmoc (const void * a, const void * b)
{
    return -1 * stonesoup_comp(a, b);
}

void stonesoup_readFile(char *filename) {
    FILE *fifo;
    char ch;

#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE,
stonesoup_readFile);
#endif /* STONESOUP_TRACE */

    fifo = fopen(filename, "r");

    if (fifo != NULL) {
        while ((ch = fgetc(fifo)) != EOF) {
            stonesoup_printf("%c", ch);
        }

        fclose(fifo);
```

Approved for public release; distribution unlimited.

```
    }
}

void *toCap (void *data) {
    struct stonesoup_data *stonesoupData = (struct stonesoup_data*)data;
    int *stonesoup_arr;
    int stonesoup_i = 0;
    int i = 0;

#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE, "toCap");
#endif /* STONESOUP_TRACE */

    stonesoup_printf("Inside toCap\n");

#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "Before sem_wait in toCap()");
#endif /* STONESOUP_TRACE */
    sem_wait(&stonesoup_sem);                     /* sem lock fails when extra unlock
occurs */
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "After sem_wait in toCap()");
#endif /* STONESOUP_TRACE */

    /* slow things down to make correct thing happen in good cases */
    stonesoup_arr = malloc(sizeof(int) * stonesoupData->qsize);
    for (stonesoup_i = 0; stonesoup_i < stonesoupData->qsize; stonesoup_i++) {
        stonesoup_arr[stonesoup_i] = stonesoupData->qsize - stonesoup_i;
    }
    qsort(stonesoup_arr, stonesoupData->qsize, sizeof(int), &stonesoup_comp);
    free(stonesoup_arr);

    stonesoup_readFile(stonesoupData->file1);

    for(i = 0; i < strlen(stonesoupData->data); i++) {
        if (stonesoupData->data[i] >= 'a' && stonesoupData->data[i] <= 'z') {      /*
null pointer dereference when concurrent */
            stonesoupData->data[i] -= 32;
/*  with other thread */
        }
    }
    sem_post(&stonesoup_sem);
    return NULL;
}

int stonesoup_isalpha(char c) {
    return ((c >= 'A' && c <= 'Z') ||
            (c >= 'a' && c <= 'z'));
}

void *delNonAlpha (void *data) {
    struct stonesoup_data *stonesoupData = (struct stonesoup_data*)data;
    int i = 0;
    int j = 0;
    char *temp = NULL;

#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE,
"delNonAlpha");
```

```
#endif /* STONESOUP_TRACE */

    stonesoup_printf("Inside delNonAlpha\n");
    /* strip all non-alpha char from global char* */
    sem_wait(&stonesoup_sem);
    temp = malloc(sizeof(char) * (strlen(stonesoupData->data) + 1));

    while(stonesoupData->data[i] != '\0') {
        if (stonesoup_isalpha(stonesoupData->data[i])) {
            temp[j++] = stonesoupData->data[i];
        }
        i++;
    }
    temp[++j] = '\0';

    free(stonesoupData->data);
    stonesoupData->data = NULL;              /* after this line, other thread runs and
dereferences null pointer */

#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
    /* STONESOUP: TRIGGER-POINT (unlockedresourceunlock) */
    stonesoup_readFile(stonesoupData->file2);
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: AFTER");
#endif /* STONESOUP_TRACE */

    stonesoupData->data = temp;
    sem_post(&stonesoup_sem);
    return NULL;
}

#endif /* STONESOUP_SNIPPET_SUPPORT */

#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 *****************************************************************************/
void weakness(char * stonesoup_taintedValue) {
#define STONESOUP_TAINT_SOURCE stonesoup_taintedValue
#endif /* STONESOUP_SNIPPET_DEBUG */

#if STONESOUP_SNIPPET_BODY_DECLARATIONS
    pthread_t stonesoup_t0, stonesoup_t1;
    int hasNonAlpha = 0;
    int stonesoup_i = 0;
    struct stonesoup_data* stonesoupData;
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */

#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
```

```
    tracepoint(stonesoup_trace, weakness_start, "CWE765", "B", "Multiple Unlocks of a
Critical Resource");
#endif /* STONESOUP_TRACE */
    stonesoupData =  malloc(sizeof(struct stonesoup_data));
    if (stonesoupData) {
        stonesoupData->data  = malloc(sizeof(char) * (strlen(STONESOUP_TAINT_SOURCE) +
1));
        stonesoupData->file1 = malloc(sizeof(char) * (strlen(STONESOUP_TAINT_SOURCE) +
1));
        stonesoupData->file2 = malloc(sizeof(char) * (strlen(STONESOUP_TAINT_SOURCE) +
1));
        if (stonesoupData->data) {
            if ((sscanf(STONESOUP_TAINT_SOURCE, "%d %s %s %s",
                      &(stonesoupData->qsize),
                        stonesoupData->file1,
                        stonesoupData->file2,
                        stonesoupData->data)  == 4) &&
                (strlen(stonesoupData->data)  != 0) &&
                (strlen(stonesoupData->file1) != 0) &&
                (strlen(stonesoupData->file2) != 0)) {

                sem_init(&stonesoup_sem, 0, 1);
                while (stonesoupData->data[stonesoup_i] != '\0') {
/* parse input for non-alpha */
                    if(stonesoup_isalpha(stonesoupData->data[stonesoup_i]) == 0) {
                        hasNonAlpha = 1;
                    }
                    stonesoup_i++;
                }

                if (hasNonAlpha != 0) {
#if STONESOUP_TRACE
                    tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT:
BEFORE");
#endif /* STONESOUP_TRACE */
                    /* STONESOUP: CROSSOVER-POINT (unlockedresourceunlock) */
                    sem_post(&stonesoup_sem);
                    pthread_create(&stonesoup_t0, NULL, delNonAlpha, stonesoupData);
/* thread will run concurrently with */
#if STONESOUP_TRACE
                    tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT:
AFTER");
#endif /* STONESOUP_TRACE */
                }                                                              /*
next thread due to the unlock on the semaphore */

                pthread_create(&stonesoup_t1, NULL, toCap, stonesoupData);

                if (hasNonAlpha != 0) {
                    pthread_join(stonesoup_t0, NULL);
                }

                pthread_join(stonesoup_t1, NULL);
            } else {
                stonesoup_printf("Error parsing data\n");
            }
            free(stonesoupData->data);
        }
        free(stonesoupData);
```

```
    }
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */

#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 *****************************************************************************/
int main(int argc,char *argv[])
{
    char* tainted_buff;
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    tainted_buff = argv[1];

    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
```

```
    return 0;
}
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-785A Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1
#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1
#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1
#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1
#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1
#endif /* NOTHING DEFINED */

#if STONESOUP_SNIPPET_INCLUDES
/******************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 ******************************************************************************/
#include <stdlib.h> /* malloc */
#include <stdio.h> /* fprintf */
#include <string.h> /* memset */
#include <unistd.h> /* getcwd */
#include <errno.h> /* errno */
/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_INCLUDES */
#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
/******************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
```

```
 ******************************************************************************/
#include "output-handlers/handler.h"
#endif /* STONESOUP_SNIPPET_DEBUG */
int stonesoup_toupper(int c)
{
  if (c >= 97 && c <= 122) {
    return c - 32;
  }
  return c;
}
#endif /* STONESOUP_SNIPPET_SUPPORT */

#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 ******************************************************************************/
void weakness(char * stonesoup_tainted_buff) {
#define STONESOUP_TAINT_SOURCE stonesoup_tainted_buff
#endif /* STONESOUP_SNIPPET_DEBUG */
#if STONESOUP_SNIPPET_BODY_DECLARATIONS
 int stonesoup_oc_i = 0;
 int stonesoup_opt_var;
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_start, "CWE785", "A", "Use of Path
Manipulation Function without Maximum-sized Buffer");
#endif /* STONESOUP_TRACE */
 size_t (*stonesoup_canary_pointer[1])(const char *);
 char stonesoup_base_path[20];
    if (strlen(STONESOUP_TAINT_SOURCE) < 20) {
        stonesoup_canary_pointer[0] = strlen;
        memset(stonesoup_base_path,0,20);
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, variable_signed_integral, "stonesoup_oc_i",
stonesoup_oc_i, &stonesoup_oc_i, "INITIAL-STATE");
        tracepoint(stonesoup_trace, variable_signed_integral, "stonesoup_opt_var",
stonesoup_opt_var, &stonesoup_opt_var, "INITIAL-STATE");
        tracepoint(stonesoup_trace, variable_address, "stonesoup_canary_pointer",
stonesoup_canary_pointer, "INITIAL-STATE");
        tracepoint(stonesoup_trace, variable_address, "stonesoup_canary_pointer[0]",
stonesoup_canary_pointer[0], "INITIAL-STATE");
        tracepoint(stonesoup_trace, variable_address, "stonesoup_base_path",
stonesoup_base_path, "INITIAL-STATE");
        tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
        tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
  /* STONESOUP: CROSSOVER-POINT (Use of Path Manipulation Function without
Maximum-sized Buffer) */
  /* STONESOUP: TRIGGER-POINT (Use of Path Manipulation Function without Maximum-sized
Buffer) */
        realpath(STONESOUP_TAINT_SOURCE,stonesoup_base_path);
```

Approved for public release; distribution unlimited.

```
        stonesoup_opt_var = stonesoup_canary_pointer[0](stonesoup_base_path);
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, variable_signed_integral, "stonesoup_opt_var",
stonesoup_opt_var, &stonesoup_opt_var, "TRIGGER-STATE");
#endif /* STONESOUP_TRACE */
        for (; stonesoup_oc_i < stonesoup_opt_var; ++stonesoup_oc_i) {
   stonesoup_base_path[stonesoup_oc_i] =
stonesoup_toupper(stonesoup_base_path[stonesoup_oc_i]);
        }
        stonesoup_printf("%s\n",stonesoup_base_path);
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: AFTER");
        tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
    }
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 ******************************************************************************/
int main(int argc,char *argv[])
{
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    char* tainted_buff = argv[1];

    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
    return 0;
```

```
}
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-785B Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1
#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1
#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1
#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1
#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1
#endif /* NOTHING DEFINED */

#if STONESOUP_SNIPPET_INCLUDES
/*****************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 *****************************************************************************/
#include <stdlib.h> /* malloc */
#include <stdio.h> /* fprintf */
#include <string.h> /* memset */
#include <errno.h> /* errno */
/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_INCLUDES */
#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 *****************************************************************************/
#include "output-handlers/handler.h"
```

```
#endif /* STONESOUP_SNIPPET_DEBUG */
int stonesoup_toupper(int c)
{
  if (c >= 97 && c <= 122) {
    return c - 32;
  }
  return c;
}
#endif /* STONESOUP_SNIPPET_SUPPORT */

#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 ******************************************************************************/
void weakness(char * stonesoup_tainted_buff) {
#define STONESOUP_TAINT_SOURCE stonesoup_tainted_buff
#endif /* STONESOUP_SNIPPET_DEBUG */
#if STONESOUP_SNIPPET_BODY_DECLARATIONS
    int stonesoup_oc_i = 0;
    int stonesoup_opt_var = 0;
    char* stonesoup_buff = 0;
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_start, "CWE785", "B", "Use of Path
Manipulation Function without Maximum-sized Buffer");
#endif /* STONESOUP_TRACE */
    if (strlen(STONESOUP_TAINT_SOURCE) < 20) {;
        stonesoup_buff = (char *) malloc (sizeof(char) * 20);
        if (stonesoup_buff != NULL) {
            memset(stonesoup_buff, 0, 20);
#if STONESOUP_TRACE
            tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
            tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
/* STONESOUP: CROSSOVER-POINT (Use of Path Manipulation Function without Maximum-sized
Buffer) */
/* STONESOUP: TRIGGER-POINT (Use of Path Manipulation Function without Maximum-sized
Buffer) */
            realpath(STONESOUP_TAINT_SOURCE, stonesoup_buff);
            stonesoup_opt_var = strlen( stonesoup_buff);
#if STONESOUP_TRACE
            tracepoint(stonesoup_trace, variable_signed_integral, "stonesoup_opt_var",
stonesoup_opt_var, &stonesoup_opt_var, "TRIGGER-STATE");
#endif /* STONESOUP_TRACE */
            for (; stonesoup_oc_i < stonesoup_opt_var; ++stonesoup_oc_i) {
                stonesoup_buff[stonesoup_oc_i] =
                    stonesoup_toupper(stonesoup_buff[stonesoup_oc_i]);
            }
            stonesoup_printf("%s\n", stonesoup_buff);
#if STONESOUP_TRACE
            tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: AFTER");
```

```
            tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
            free (stonesoup_buff);
        }
    }
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 *******************************************************************************/
int main(int argc,char *argv[])
{
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    char* tainted_buff = argv[1];

    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
    return 0;
```

```
}
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-785C Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1
#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1
#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1
#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1
#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1
#endif /* NOTHING DEFINED */

#if STONESOUP_SNIPPET_INCLUDES
/******************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 ******************************************************************************/
#include <stdlib.h> /* malloc */
#include <stdio.h> /* fprintf */
#include <string.h> /* memset */
#include <errno.h> /* errno */
/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_INCLUDES */
#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
/******************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 ******************************************************************************/
#include "output-handlers/handler.h"
```

```
#endif /* STONESOUP_SNIPPET_DEBUG */
struct stonesoup_struct {
    char base_path[20];
    char * buff_pointer;
};

int stonesoup_toupper(int c)
{
  if (c >= 97 && c <= 122) {
    return c - 32;
  }
  return c;
}
#endif /* STONESOUP_SNIPPET_SUPPORT */

#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 ******************************************************************************/
void weakness(char * stonesoup_tainted_buff) {
#define STONESOUP_TAINT_SOURCE stonesoup_tainted_buff
#endif /* STONESOUP_SNIPPET_DEBUG */
#if STONESOUP_SNIPPET_BODY_DECLARATIONS
    int stonesoup_oc_i = 0;
    int stonesoup_opt_var = 0;
    struct stonesoup_struct* stonesoup_data = 0;
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_start, "CWE785", "C", "Use of Path
Manipulation Function without Maximum-sized Buffer");
#endif /* STONESOUP_TRACE */
    if (strlen(STONESOUP_TAINT_SOURCE) < 20) {;
        stonesoup_data = (struct stonesoup_struct*) malloc (sizeof(struct
stonesoup_struct));
        if (stonesoup_data != NULL) {
            memset(stonesoup_data->base_path, 0, 20);
            stonesoup_data->buff_pointer = stonesoup_data->base_path;
#if STONESOUP_TRACE
            tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
            tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
/* STONESOUP: CROSSOVER-POINT (Use of Path Manipulation Function without Maximum-sized
Buffer) */
/* STONESOUP: TRIGGER-POINT (Use of Path Manipulation Function without Maximum-sized
Buffer) */
            realpath(STONESOUP_TAINT_SOURCE, stonesoup_data->base_path);
            stonesoup_opt_var = strlen( stonesoup_data->buff_pointer);
#if STONESOUP_TRACE
            tracepoint(stonesoup_trace, variable_signed_integral, "stonesoup_opt_var",
stonesoup_opt_var, &stonesoup_opt_var, "TRIGGER-POINT");
#endif /* STONESOUP_TRACE */
```

```
            for (; stonesoup_oc_i < stonesoup_opt_var; ++stonesoup_oc_i) {
                stonesoup_data->base_path[stonesoup_oc_i] =
                    stonesoup_toupper(stonesoup_data->base_path[stonesoup_oc_i]);
            }
            stonesoup_printf("%s\n", stonesoup_data->base_path);
            stonesoup_printf("%s\n", stonesoup_data->buff_pointer);
#if STONESOUP_TRACE
            tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: AFTER");
            tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
            free (stonesoup_data);
        }
    }
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_DEBUG
/******************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 ******************************************************************************/
int main(int argc,char *argv[])
{
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    char* tainted_buff = argv[1];

    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
    return 0;
```

```
}
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-785D Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1
#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1
#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1
#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1
#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1
#endif /* NOTHING DEFINED */

#if STONESOUP_SNIPPET_INCLUDES
/*****************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 *****************************************************************************/
#include <stdlib.h> /* malloc */
#include <stdio.h> /* fprintf */
#include <string.h> /* memset */
#include <errno.h> /* errno */
/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_INCLUDES */
#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 *****************************************************************************/
#include "output-handlers/handler.h"
```

```
#endif /* STONESOUP_SNIPPET_DEBUG */
struct stonesoup_struct {
    char base_path[20];
    char * buff_pointer;
};

int stonesoup_toupper(int c)
{
  if (c >= 97 && c <= 122) {
    return c - 32;
  }
  return c;
}
#endif /* STONESOUP_SNIPPET_SUPPORT */

#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 ******************************************************************************/
void weakness(char * stonesoup_tainted_buff) {
#define STONESOUP_TAINT_SOURCE stonesoup_tainted_buff
#endif /* STONESOUP_SNIPPET_DEBUG */
#if STONESOUP_SNIPPET_BODY_DECLARATIONS
    int stonesoup_oc_i = 0;
    int stonesoup_opt_var = 0;
    int stonesoup_i;
    struct stonesoup_struct stonesoup_data;
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_start, "CWE785", "D", "Use of Path
Manipulation Function without Maximum-sized Buffer");
#endif /* STONESOUP_TRACE */
    if (strlen(STONESOUP_TAINT_SOURCE) < 20) {
        for (stonesoup_i = 0; stonesoup_i < 20; stonesoup_i++) {
            stonesoup_data.base_path[stonesoup_i] = 0;
        }
        stonesoup_data.buff_pointer = stonesoup_data.base_path;
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
        tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
/* STONESOUP: CROSSOVER-POINT (Use of Path Manipulation Function without Maximum-sized
Buffer) */
/* STONESOUP: TRIGGER-POINT (Use of Path Manipulation Function without Maximum-sized
Buffer) */
        realpath(STONESOUP_TAINT_SOURCE, stonesoup_data.base_path);
        stonesoup_opt_var = strlen( stonesoup_data.buff_pointer);
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, variable_signed_integral, "stonesoup_opt_var",
stonesoup_opt_var, &stonesoup_opt_var, "TRIGGER-STATE");
#endif /* STONESOUP_TRACE */
```

```
        for (; stonesoup_oc_i < stonesoup_opt_var; ++stonesoup_oc_i) {
            stonesoup_data.base_path[stonesoup_oc_i] =
                stonesoup_toupper(stonesoup_data.base_path[stonesoup_oc_i]);
        }
        stonesoup_printf("%s\n", stonesoup_data.base_path);
        stonesoup_printf("%s\n", stonesoup_data.buff_pointer);
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: AFTER");
        tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
    }
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 ******************************************************************************/
int main(int argc,char *argv[])
{
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    char* tainted_buff = argv[1];

    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
    return 0;
```

```
}
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-805A Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1
#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1
#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1
#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1
#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1
#endif /* NOTHING DEFINED */

#if STONESOUP_SNIPPET_INCLUDES
/*****************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 *****************************************************************************/
#include <stdlib.h> /* malloc */
#include <stdio.h> /* fprintf */
#include <string.h> /* memset */
#include <errno.h> /* errno */
/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_INCLUDES */
#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 *****************************************************************************/
#include "output-handlers/handler.h"
```

```
#endif /* STONESOUP_SNIPPET_DEBUG */
int stonesoup_toupper(int c)
{
  if (c >= 97 && c <= 122) {
    return c - 32;
  }
  return c;
}
#endif /* STONESOUP_SNIPPET_SUPPORT */

#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 *******************************************************************************/
void weakness(char * stonesoup_tainted_buff) {
#define STONESOUP_TAINT_SOURCE stonesoup_tainted_buff
#endif /* STONESOUP_SNIPPET_DEBUG */
#if STONESOUP_SNIPPET_BODY_DECLARATIONS
 int stonesoup_oc_i = 0;
 char stonesoup_buffer[8];
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_start, "CWE805", "A", "Buffer Access with
Incorrect Length Value");
    tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
    tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
 /* STONESOUP: CROSSOVER-POINT (Buffer Access With Incorrect Length Value) */
 /* STONESOUP: TRIGGER-POINT (Buffer Access With Incorrect Length Value) */
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, variable_buffer_info, "STONESOUP_TAINT_SOURCE",
strlen(STONESOUP_TAINT_SOURCE)+1, STONESOUP_TAINT_SOURCE, "TRIGGER-STATE");
#endif /* STONESOUP_TRACE */
    strncpy(stonesoup_buffer,STONESOUP_TAINT_SOURCE,strlen(STONESOUP_TAINT_SOURCE) +
1);
    for (; stonesoup_oc_i < strlen(stonesoup_buffer); ++stonesoup_oc_i) {
        stonesoup_buffer[stonesoup_oc_i] =
stonesoup_toupper(stonesoup_buffer[stonesoup_oc_i]);
    }
    stonesoup_printf("%s\n",stonesoup_buffer);
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: AFTER");
    tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
    tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_DEBUG
```

```
/*******************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 *******************************************************************************/
int main(int argc,char *argv[])
{
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    char* tainted_buff = argv[1];

    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
    return 0;
```

Approved for public release; distribution unlimited.

```
}
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-805B Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1
#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1
#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1
#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1
#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1
#endif /* NOTHING DEFINED */

#if STONESOUP_SNIPPET_INCLUDES
/*****************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 *****************************************************************************/
#include <stdlib.h> /* malloc */
#include <stdio.h> /* fprintf */
#include <string.h> /* memset */
#include <errno.h> /* errno */
/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_INCLUDES */
#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 *****************************************************************************/
#include "output-handlers/handler.h"
```

```
#endif /* STONESOUP_SNIPPET_DEBUG */
int stonesoup_toupper(int c)
{
  if (c >= 97 && c <= 122) {
    return c - 32;
  }
  return c;
}
#endif /* STONESOUP_SNIPPET_SUPPORT */

#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/******************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 ******************************************************************************/
void weakness(char * stonesoup_tainted_buff) {
#define STONESOUP_TAINT_SOURCE stonesoup_tainted_buff
#endif /* STONESOUP_SNIPPET_DEBUG */
#if STONESOUP_SNIPPET_BODY_DECLARATIONS
    int stonesoup_oc_i = 0;
    char * stonesoup_data = 0;
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_start, "CWE805", "B", "Buffer Access with
Incorrect Length Value");
#endif /* STONESOUP_TRACE */
    stonesoup_data = (char*) malloc(8 * sizeof(char));
    if (stonesoup_data != NULL) {
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
        tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
        /* STONESOUP: CROSSOVER-POINT (Buffer Access With Incorrect Length Value) */
        /* STONESOUP: TRIGGER-POINT (Buffer Access With Incorrect Length Value) */
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, variable_buffer_info, "STONESOUP_TAINT_SOURCE",
strlen(STONESOUP_TAINT_SOURCE)+1, STONESOUP_TAINT_SOURCE, "TRIGGER-STATE");
#endif /* STONESOUP_TRACE */
        strncpy(stonesoup_data, STONESOUP_TAINT_SOURCE, strlen(STONESOUP_TAINT_SOURCE)
+ 1);
        for (; stonesoup_oc_i < strlen(stonesoup_data); ++stonesoup_oc_i) {
            stonesoup_data[stonesoup_oc_i] =
stonesoup_toupper(stonesoup_data[stonesoup_oc_i]);
        }
        stonesoup_printf("%s\n", stonesoup_data);
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: AFTER");
        tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
        free(stonesoup_data);
    }
#if STONESOUP_TRACE
```

```
    tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 *****************************************************************************/
int main(int argc,char *argv[])
{
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    char* tainted_buff = argv[1];

    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
    return 0;
```

```
}
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-805C Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1
#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1
#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1
#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1
#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1
#endif /* NOTHING DEFINED */

#if STONESOUP_SNIPPET_INCLUDES
/*****************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 *****************************************************************************/
#include <stdlib.h> /* malloc */
#include <stdio.h> /* fprintf */
#include <string.h> /* memset */
#include <errno.h> /* errno */
/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_INCLUDES */
#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 *****************************************************************************/
#include "output-handlers/handler.h"
```

```
#endif /* STONESOUP_SNIPPET_DEBUG */
int stonesoup_toupper(int c)
{
  if (c >= 97 && c <= 122) {
    return c - 32;
  }
  return c;
}
struct stonesoup_struct {
    char buffer[8];
    char * buff_pointer;
};
#endif /* STONESOUP_SNIPPET_SUPPORT */

#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 ******************************************************************************/
void weakness(char * stonesoup_tainted_buff) {
#define STONESOUP_TAINT_SOURCE stonesoup_tainted_buff
#endif /* STONESOUP_SNIPPET_DEBUG */
#if STONESOUP_SNIPPET_BODY_DECLARATIONS
    int stonesoup_oc_i = 0;
    int stonesoup_ptr_deref;
    struct stonesoup_struct * stonesoup_data = 0;
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_start, "CWE805", "C", "Buffer Access with
Incorrect Length Value");
#endif /* STONESOUP_TRACE */
    stonesoup_data = (struct stonesoup_struct*) malloc(sizeof (struct
stonesoup_struct));
    if (stonesoup_data != NULL) {
        stonesoup_data->buff_pointer = stonesoup_data->buffer;
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
        tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
        /* STONESOUP: CROSSOVER-POINT (Buffer Access With Incorrect Length Value) */
        /* STONESOUP: TRIGGER-POINT (Buffer Access With Incorrect Length Value) */
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, variable_buffer_info, "STONESOUP_TAINT_SOURCE",
strlen(STONESOUP_TAINT_SOURCE)+1, STONESOUP_TAINT_SOURCE, "TAINTED-SOURCE");
#endif /* STONESOUP_TRACE */
        strncpy(stonesoup_data->buffer, STONESOUP_TAINT_SOURCE,
strlen(STONESOUP_TAINT_SOURCE) + 1);
        stonesoup_ptr_deref = strlen( stonesoup_data->buff_pointer);
        for (; stonesoup_oc_i < stonesoup_ptr_deref; ++stonesoup_oc_i) {
            stonesoup_data->buffer[stonesoup_oc_i] =
stonesoup_toupper(stonesoup_data->buffer[stonesoup_oc_i]);
        }
```

```
        stonesoup_printf("%s\n", stonesoup_data->buffer);
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: AFTER");
        tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
        free(stonesoup_data);
    }
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 ******************************************************************************/
int main(int argc,char *argv[])
{
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    char* tainted_buff = argv[1];

    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
    return 0;
```

```
}
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-805D Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1
#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1
#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1
#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1
#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1
#endif /* NOTHING DEFINED */

#if STONESOUP_SNIPPET_INCLUDES
/******************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 ******************************************************************************/
#include <stdlib.h> /* malloc */
#include <stdio.h> /* fprintf */
#include <string.h> /* memset */
#include <errno.h> /* errno */
/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_INCLUDES */
#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
/******************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 ******************************************************************************/
#include "output-handlers/handler.h"
```

```
#endif /* STONESOUP_SNIPPET_DEBUG */
int stonesoup_toupper(int c)
{
  if (c >= 97 && c <= 122) {
    return c - 32;
  }
  return c;
}
struct stonesoup_struct {
    char buffer[8];
    char * buff_pointer;
};
#endif /* STONESOUP_SNIPPET_SUPPORT */

#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 ******************************************************************************/
void weakness(char * stonesoup_tainted_buff) {
#define STONESOUP_TAINT_SOURCE stonesoup_tainted_buff
#endif /* STONESOUP_SNIPPET_DEBUG */
#if STONESOUP_SNIPPET_BODY_DECLARATIONS
    int stonesoup_oc_i = 0;
    int stonesoup_ptr_deref;
    struct stonesoup_struct stonesoup_data;
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_start, "CWE805", "D", "Buffer Access with
Incorrect Length Value");
#endif /* STONESOUP_TRACE */
    stonesoup_data.buff_pointer = stonesoup_data.buffer;
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
    tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
    /* STONESOUP: CROSSOVER-POINT (Buffer Access With Incorrect Length Value) */
    /* STONESOUP: TRIGGER-POINT (Buffer Access With Incorrect Length Value) */
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, variable_buffer_info, "STONESOUP_TAINT_SOURCE",
strlen(STONESOUP_TAINT_SOURCE)+1, STONESOUP_TAINT_SOURCE, "TRIGGER-STATE");
#endif /* STONESOUP_TRACE */
    strncpy(stonesoup_data.buffer, STONESOUP_TAINT_SOURCE,
strlen(STONESOUP_TAINT_SOURCE) + 1);
    stonesoup_ptr_deref = strlen( stonesoup_data.buff_pointer);
    for (; stonesoup_oc_i < stonesoup_ptr_deref; ++stonesoup_oc_i) {
        stonesoup_data.buffer[stonesoup_oc_i] =
stonesoup_toupper(stonesoup_data.buffer[stonesoup_oc_i]);
    }
    stonesoup_printf("%s\n", stonesoup_data.buffer);
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: AFTER");
```

```
    tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
    tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_DEBUG
/******************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 ******************************************************************************/
int main(int argc,char *argv[])
{
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    char* tainted_buff = argv[1];

    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
    return 0;
```

```
}
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-806A Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1
#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1
#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1
#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1
#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1
#endif /* NOTHING DEFINED */

#if STONESOUP_SNIPPET_INCLUDES
/******************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 ******************************************************************************/
#include <stdlib.h> /* malloc */
#include <stdio.h> /* fprintf */
#include <string.h> /* memset */
#include <errno.h> /* errno */
/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_INCLUDES */
#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
/******************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 ******************************************************************************/
#include "output-handlers/handler.h"
```

```
#endif /* STONESOUP_SNIPPET_DEBUG */
int stonesoup_toupper(int c)
{
  if (c >= 97 && c <= 122) {
    return c - 32;
  }
  return c;
}
#endif /* STONESOUP_SNIPPET_SUPPORT */

#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 ******************************************************************************/
void weakness(char * stonesoup_tainted_buff) {
#define STONESOUP_TAINT_SOURCE stonesoup_tainted_buff
#endif /* STONESOUP_SNIPPET_DEBUG */
#if STONESOUP_SNIPPET_BODY_DECLARATIONS
 int stonesoup_oc_i = 0;
 char stonesoup_buff[64];
 char stonesoup_source[1024];
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_start, "CWE806", "A", "Buffer Access Using
Size of Source Buffer");
#endif /* STONESOUP_TRACE */
    memset(stonesoup_buff, 65, 64);
    stonesoup_buff[63] = '\0';
    memset(stonesoup_source,0,1024);
    strncpy(stonesoup_source,STONESOUP_TAINT_SOURCE,sizeof(stonesoup_source));
    stonesoup_source[1023] = 0;
    if (strlen(stonesoup_source) + 1 <= sizeof(stonesoup_buff))  {
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
        tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
        /* STONESOUP: CROSSOVER-POINT (Buffer Access Using Size of Source Buffer) */
     /* STONESOUP: TRIGGER-POINT (Buffer Access Using Size of Source Buffer) */
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, variable_buffer_info, "stonesoup_source",
strlen(stonesoup_source)+1, stonesoup_source, "TRIGGER-STATE");
        tracepoint(stonesoup_trace, variable_buffer_info, "stonesoup_buff",
strlen(stonesoup_buff)+1, stonesoup_buff, "TRIGGER-STATE");
#endif /* STONESOUP_TRACE */
        strncpy(stonesoup_buff,stonesoup_source,sizeof(stonesoup_source));
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: AFTER");
        tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
    }
    for (; stonesoup_oc_i < strlen(stonesoup_buff); ++stonesoup_oc_i) {
```

```
        stonesoup_buff[stonesoup_oc_i] =
stonesoup_toupper(stonesoup_buff[stonesoup_oc_i]);
    }
    stonesoup_printf("%s\n",stonesoup_buff);
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 *****************************************************************************/
int main(int argc,char *argv[])
{
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    char* tainted_buff = argv[1];

    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
    return 0;
```

```
}
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-806B Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1
#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1
#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1
#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1
#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1
#endif /* NOTHING DEFINED */

#if STONESOUP_SNIPPET_INCLUDES
/*****************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 *****************************************************************************/
#include <stdlib.h> /* malloc */
#include <stdio.h> /* fprintf */
#include <string.h> /* memset */
#include <errno.h> /* errno */
/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_INCLUDES */
#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 *****************************************************************************/
#include "output-handlers/handler.h"
```

```
#endif /* STONESOUP_SNIPPET_DEBUG */
int stonesoup_toupper(int c)
{
  if (c >= 97 && c <= 122) {
    return c - 32;
  }
  return c;
}
#endif /* STONESOUP_SNIPPET_SUPPORT */

#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 *****************************************************************************/
void weakness(char * stonesoup_tainted_buff) {
#define STONESOUP_TAINT_SOURCE stonesoup_tainted_buff
#endif /* STONESOUP_SNIPPET_DEBUG */
#if STONESOUP_SNIPPET_BODY_DECLARATIONS
    int stonesoup_oc_i = 0;
    int stonesoup_opt_var;
    char stonesoup_source[1024];
    char * stonesoup_buffer;
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_start, "CWE806", "B", "Buffer Access Using
Size of Source Buffer");
#endif /* STONESOUP_TRACE */
    stonesoup_buffer = (char*) malloc (sizeof(char*) * 64);
    if (stonesoup_buffer != NULL) {
        memset(stonesoup_source, 0, 1024);
        memset(stonesoup_buffer, 65, 64);
        stonesoup_buffer[64 - 1] = '\0';
        strncpy(stonesoup_source, STONESOUP_TAINT_SOURCE, sizeof(stonesoup_source));
        stonesoup_source[1023] = '\0';
        if (strlen(stonesoup_source) + 1 <= 64) {
#if STONESOUP_TRACE
            tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
            tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
            /* STONESOUP: CROSSOVER-POINT (Buffer Access Using Size of Source Buffer)
*/
            /* STONESOUP: TRIGGER-POINT (Buffer Access Using Size of Source Buffer) */

#if STONESOUP_TRACE
            tracepoint(stonesoup_trace, variable_buffer_info, "stonesoup_source",
strlen(stonesoup_source)+1, stonesoup_source, "TRIGGER-STATE");
            tracepoint(stonesoup_trace, variable_buffer_info, "stonesoup_buffer",
strlen(stonesoup_buffer)+1, stonesoup_buffer, "TRIGGER-STATE");
#endif /* STONESOUP_TRACE */
            strncpy(stonesoup_buffer, stonesoup_source, sizeof(stonesoup_source));
#if STONESOUP_TRACE
```

```
            tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: AFTER");
            tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
        }
        stonesoup_opt_var = strlen(stonesoup_buffer);
        for (; stonesoup_oc_i < stonesoup_opt_var; ++stonesoup_oc_i) {
            stonesoup_buffer[stonesoup_oc_i] =
                stonesoup_toupper(stonesoup_buffer[stonesoup_oc_i]);
        }
        stonesoup_printf("%s\n", stonesoup_buffer);
        free(stonesoup_buffer);
    }
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 *****************************************************************************/
int main(int argc,char *argv[])
{
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    char* tainted_buff = argv[1];

    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
    return 0;
```

```
}
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-806C Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1
#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1
#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1
#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1
#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1
#endif /* NOTHING DEFINED */

#if STONESOUP_SNIPPET_INCLUDES
/******************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 ******************************************************************************/
#include <stdlib.h> /* malloc */
#include <stdio.h> /* fprintf */
#include <string.h> /* memset */
#include <errno.h> /* errno */
/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_INCLUDES */
#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
/******************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 ******************************************************************************/
#include "output-handlers/handler.h"
```

```
#endif /* STONESOUP_SNIPPET_DEBUG */
int stonesoup_toupper(int c) {
  if (c >= 97 && c <= 122) {
    return c - 32;
  }
  return c;
}
struct stonesoup_struct {
    char buffer[64];
    char * buff_pointer;
};
#endif /* STONESOUP_SNIPPET_SUPPORT */

#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 *****************************************************************************/
void weakness(char * stonesoup_tainted_buff) {
#define STONESOUP_TAINT_SOURCE stonesoup_tainted_buff
#endif /* STONESOUP_SNIPPET_DEBUG */
#if STONESOUP_SNIPPET_BODY_DECLARATIONS
    int stonesoup_oc_i = 0;
    int stonesoup_opt_var;
    char stonesoup_source[1024];
    struct stonesoup_struct * stonesoup_data = 0;
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_start, "CWE806", "C", "Buffer Access Using
Size of Source Buffer");
#endif /* STONESOUP_TRACE */
    stonesoup_data = (struct stonesoup_struct *) malloc (sizeof(struct
stonesoup_struct));
    if (stonesoup_data != NULL) {
        memset(stonesoup_source, 0, 1024);
        memset(stonesoup_data->buffer, 65, 64);
        stonesoup_data->buffer[64 - 1] = '\0';
        stonesoup_data->buff_pointer = stonesoup_data->buffer;

        strncpy(stonesoup_source, STONESOUP_TAINT_SOURCE, sizeof(stonesoup_source));
        stonesoup_source[1023] = '\0';
        if (strlen(stonesoup_source) + 1 <= 64) {
#if STONESOUP_TRACE
            tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
            tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
            /* STONESOUP: CROSSOVER-POINT (Buffer Access Using Size of Source Buffer)
*/
            /* STONESOUP: TRIGGER-POINT (Buffer Access Using Size of Source Buffer) */
#if STONESOUP_TRACE
            tracepoint(stonesoup_trace, variable_buffer_info, "stonesoup_source",
strlen(stonesoup_source)+1, stonesoup_source, "TRIGGER-STATE");
```

```
            tracepoint(stonesoup_trace, variable_buffer_info,
"stonesoup_data->buffer", strlen(stonesoup_data->buffer)+1, stonesoup_data->buffer,
"TRIGGER-STATE");
#endif /* STONESOUP_TRACE */
            strncpy(stonesoup_data->buffer, stonesoup_source,
sizeof(stonesoup_source));
#if STONESOUP_TRACE
            tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: AFTER");
            tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
        }
        stonesoup_opt_var = strlen( stonesoup_data->buff_pointer);
        for (; stonesoup_oc_i < stonesoup_opt_var; ++stonesoup_oc_i) {
            stonesoup_data->buffer[stonesoup_oc_i] =
                stonesoup_toupper(stonesoup_data->buffer[stonesoup_oc_i]);
        }
        stonesoup_printf("%s\n", stonesoup_data->buffer);
        free(stonesoup_data);
    }
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 ******************************************************************************/
int main(int argc,char *argv[])
{
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    char* tainted_buff = argv[1];

    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
    return 0;
```

```
}
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-806D Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1
#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1
#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1
#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1
#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1
#endif /* NOTHING DEFINED */

#if STONESOUP_SNIPPET_INCLUDES
/******************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 ******************************************************************************/
#include <stdlib.h> /* malloc */
#include <stdio.h> /* fprintf */
#include <string.h> /* memset */
#include <errno.h> /* errno */
/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_INCLUDES */
#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
/******************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 ******************************************************************************/
#include "output-handlers/handler.h"
```

Approved for public release; distribution unlimited.

```
#endif /* STONESOUP_SNIPPET_DEBUG */
int stonesoup_toupper(int c) {
  if (c >= 97 && c <= 122) {
    return c - 32;
  }
  return c;
}
struct stonesoup_struct {
    char buffer[64];
    char * buff_pointer;
};
#endif /* STONESOUP_SNIPPET_SUPPORT */

#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 ******************************************************************************/
void weakness(char * stonesoup_tainted_buff) {
#define STONESOUP_TAINT_SOURCE stonesoup_tainted_buff
#endif /* STONESOUP_SNIPPET_DEBUG */
#if STONESOUP_SNIPPET_BODY_DECLARATIONS
    int stonesoup_oc_i = 0;
    int stonesoup_i;
    int stonesoup_opt_var;
    char stonesoup_source[1024];
    struct stonesoup_struct stonesoup_data;
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_start, "CWE806", "D", "Buffer Access Using
Size of Source Buffer");
#endif /* STONESOUP_TRACE */
    for (stonesoup_i = 0; stonesoup_i < 1024; stonesoup_i++) {
        stonesoup_source[stonesoup_i] = 0;
    }
    for (stonesoup_i = 0; stonesoup_i < 64; stonesoup_i++) {
        stonesoup_data.buffer[stonesoup_i] = 65;
    }
    stonesoup_data.buffer[64 - 1] = '\0';
    stonesoup_data.buff_pointer = stonesoup_data.buffer;

    strncpy(stonesoup_source, STONESOUP_TAINT_SOURCE, sizeof(stonesoup_source));
    stonesoup_source[1023] = '\0';
    if (strlen(stonesoup_source) + 1 <= 64) {
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
        tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
        /* STONESOUP: CROSSOVER-POINT (Buffer Access Using Size of Source Buffer) */
        /* STONESOUP: TRIGGER-POINT (Buffer Access Using Size of Source Buffer) */
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, variable_buffer_info, "stonesoup_source",
```

```
strlen(stonesoup_source)+1, stonesoup_source, "TRIGGER-STATE");
        tracepoint(stonesoup_trace, variable_buffer_info, "stonesoup_data.buffer",
strlen(stonesoup_data.buffer)+1, stonesoup_data.buffer, "TRIGGER-STATE");
#endif /* STONESOUP_TRACE */
        strncpy(stonesoup_data.buffer, stonesoup_source, sizeof(stonesoup_source));
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: AFTER");
        tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
    }
    stonesoup_opt_var = strlen( stonesoup_data.buff_pointer);
    for (; stonesoup_oc_i < stonesoup_opt_var; ++stonesoup_oc_i) {
        stonesoup_data.buffer[stonesoup_oc_i] =
            stonesoup_toupper(stonesoup_data.buffer[stonesoup_oc_i]);
    }
    stonesoup_printf("%s\n", stonesoup_data.buffer);
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 *******************************************************************************/
int main(int argc,char *argv[])
{
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    char* tainted_buff = argv[1];

    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
    return 0;
```

```
}
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-820A Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1

#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1

#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1

#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1

#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1

#endif /* NOTHING DEFINED */


#if STONESOUP_SNIPPET_INCLUDES
/******************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 ******************************************************************************/
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <string.h>

/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */

#endif /* STONESOUP_SNIPPET_INCLUDES */

#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
/******************************************************************************
```

```
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 ******************************************************************************/
#include "output-handlers/handler.h"

#endif /* STONESOUP_SNIPPET_DEBUG */

struct stonesoup_data {
    int inc_amount;
    int qsize;
    char *data;
    char *file1;
    char *file2;
};

int stonesoup_comp (const void * a, const void * b) {
    if (a > b) {
        return -1;
    }
    else if (a < b) {
        return 1;
    }
    else {
        return 0;
    }
}

int stonesoup_pmoc (const void * a, const void * b) {
    return -1 * stonesoup_comp(a, b);
}

void stonesoup_readFile(char *filename) {
    FILE *fifo;
    char ch;

#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE,
"stonesoup_readFile");
#endif /* STONESOUP_TRACE */

    fifo = fopen(filename, "r");

    if (fifo != NULL) {
        while ((ch = fgetc(fifo)) != EOF) {
            stonesoup_printf("%c", ch);
        }

        fclose(fifo);
    }
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "Finished reading sync file.");
#endif /* STONESOUP_TRACE */
}
```

```
void *calcIncamount(void *data) {
    struct stonesoup_data *dataStruct = (struct stonesoup_data*)data;
    stonesoup_printf("In calcInamount\n");

#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE,
"calcIncamount");
    tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */

    /* STONESOUP: CROSSOVER-POINT (missing syncronization) */
    dataStruct->inc_amount = dataStruct->data[0] - 'A';                     /*
oops...um... */

#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, variable_signed_integral, "dataStruct->inc_amount",
dataStruct->inc_amount, &dataStruct->inc_amount, "CROSSOVER-STATE");
#endif /* STONESOUP_TRACE */

    stonesoup_readFile(dataStruct->file2);

    if (dataStruct->inc_amount < 0) {                                       /* let's
just clean up and */
        dataStruct->inc_amount *= -1;                                       /*
pretend that never happened */
    }
    else if (dataStruct->inc_amount == 0) {                                 /*  shhhh
*/
        dataStruct->inc_amount += 1;
    }

#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
    tracepoint(stonesoup_trace, variable_signed_integral, "dataStruct->inc_amount",
dataStruct->inc_amount, &dataStruct->inc_amount, "FINAL-STATE");
#endif /* STONESOUP_TRACE */

    return NULL;
}

void *toPound(void *data) {
    int stonesoup_i;
    struct stonesoup_data *dataStruct = (struct stonesoup_data*)data;
    int *stonesoup_arr = NULL;

#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE, "toPound");
#endif /* STONESOUP_TRACE */

    stonesoup_printf("In toPound\n");

    /* slow things down to make correct thing happen in good cases */
    stonesoup_arr = malloc(sizeof(int) * dataStruct->qsize);
    for (stonesoup_i = 0; stonesoup_i < dataStruct->qsize; stonesoup_i++) {
        stonesoup_arr[stonesoup_i] = dataStruct->qsize - stonesoup_i;
    }
    qsort(stonesoup_arr, dataStruct->qsize, sizeof(int), &stonesoup_comp);
    free(stonesoup_arr);
```

```
    stonesoup_readFile(dataStruct->file1);

#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
    tracepoint(stonesoup_trace, variable_signed_integral, "dataStruct->inc_amount",
dataStruct->inc_amount, &dataStruct->inc_amount, "TRIGGER-STATE");
#endif /* STONESOUP_TRACE */

    /* STONESOUP: TRIGGER-POINT (missing syncronization) */
    for (stonesoup_i = 0; stonesoup_i < (int)strlen(dataStruct->data) - 1;
         stonesoup_i += dataStruct->inc_amount)                                /* can
cause underread/write if */
    {
        dataStruct->data[stonesoup_i] = '#';                                   /*
stonesoup_increment_amount is neg */
    }

#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: AFTER");
#endif /* STONESOUP_TRACE */

    return NULL;
}


#endif /* STONESOUP_SNIPPET_SUPPORT */

#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/******************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 ******************************************************************************/
void weakness(char * stonesoup_taintedValue) {
#define STONESOUP_TAINT_SOURCE stonesoup_taintedValue
#endif /* STONESOUP_SNIPPET_DEBUG */

#if STONESOUP_SNIPPET_BODY_DECLARATIONS
    pthread_t stonesoup_t0, stonesoup_t1;
    struct stonesoup_data *dataStruct = malloc(sizeof(struct stonesoup_data));
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */

#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_start, "CWE820", "A", "Missing
Synchronization");
#endif /* STONESOUP_TRACE */

    if (dataStruct) {
        dataStruct->inc_amount = 1;

        dataStruct->data  = malloc(sizeof(char) * (strlen(STONESOUP_TAINT_SOURCE) +
1));
        dataStruct->file1 = malloc(sizeof(char) * (strlen(STONESOUP_TAINT_SOURCE) +
```

```
1));
        dataStruct->file2 = malloc(sizeof(char) * (strlen(STONESOUP_TAINT_SOURCE) +
1));
        if (dataStruct->data) {
            if ((sscanf(STONESOUP_TAINT_SOURCE, "%d %s %s %s",
                      &(dataStruct->qsize),
                        dataStruct->file1,
                        dataStruct->file2,
                        dataStruct->data)  == 4) &&
                (strlen(dataStruct->data)  != 0) &&
                (strlen(dataStruct->file1) != 0) &&
                (strlen(dataStruct->file2) != 0)) {

#if STONESOUP_TRACE
                tracepoint(stonesoup_trace, variable_signed_integral,
"stonesoupData->qsize", dataStruct->qsize, &(dataStruct->qsize), "INITIAL-STATE");
                tracepoint(stonesoup_trace, variable_buffer, "stonesoupData->data",
dataStruct->data, "INITIAL-STATE");
                tracepoint(stonesoup_trace, variable_buffer, "stonesoupData->file1",
dataStruct->file1, "INITIAL-STATE");
                tracepoint(stonesoup_trace, variable_buffer, "stonesoupData->file2",
dataStruct->file2, "INITIAL-STATE");
                tracepoint(stonesoup_trace, trace_point, "Spawning threads.");
#endif /* STONESOUP_TRACE */

                if (pthread_create(&stonesoup_t0, NULL, calcIncamount,
(void*)(dataStruct)) != 0) {
                    stonesoup_printf("Error initializing thread 0.");
                }

                if (pthread_create(&stonesoup_t1, NULL, toPound, (void*)(dataStruct))
!= 0) {
                    stonesoup_printf("Error initializing thread 1.");
                }

                pthread_join(stonesoup_t0, NULL);
                pthread_join(stonesoup_t1, NULL);

#if STONESOUP_TRACE
                tracepoint(stonesoup_trace, trace_point, "Threads joined.");
#endif /* STONESOUP_TRACE */
            }

            free(dataStruct->data);
        } else {
#if STONESOUP_TRACE
                tracepoint(stonesoup_trace, trace_error, "Error parsing data.");
#endif /* STONESOUP_TRACE */
                stonesoup_printf("Error parsing data.\n");
        }
        free (dataStruct);
    } else {
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, trace_error, "Error malloc()ing space for
struct.");
#endif /* STONESOUP_TRACE */
        stonesoup_printf("Error malloc()ing space for struct.\n");
    }
```

```
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */

#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 ******************************************************************************/
int main(int argc,char *argv[])
{
    char* tainted_buff;
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    tainted_buff = argv[1];

    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
```

```
    return 0;
}
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-821A Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1

#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1

#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1

#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1

#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1

#endif /* NOTHING DEFINED */


#if STONESOUP_SNIPPET_INCLUDES
/******************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 ******************************************************************************/
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <string.h>

/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */

#endif /* STONESOUP_SNIPPET_INCLUDES */

#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
```

```
/*******************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 ******************************************************************************/
#include "output-handlers/handler.h"

#endif /* STONESOUP_SNIPPET_DEBUG */

struct stonesoup_data {
    int qsize;
    char *data;
    char *file1;
    char *file2;
};

pthread_t stonesoup_t0, stonesoup_t1;
pthread_mutex_t stonesoup_mutex_0, stonesoup_mutex_1;
int stonesoup_dev_amount = 1;

int stonesoup_comp (const void * a, const void * b) {
    if (a > b) {
        return -1;
    }
    else if (a < b) {
        return 1;
    }
    else {
        return 0;
    }
}

int stonesoup_pmoc (const void * a, const void * b) {
    return -1 * stonesoup_comp(a, b);
}

void stonesoup_readFile(char *filename) {
    FILE *fifo;
    char ch;

#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE,
"stonesoup_readFile");
#endif /* STONESOUP_TRACE */

    fifo = fopen(filename, "r");

    if (fifo != NULL) {
        while ((ch = fgetc(fifo)) != EOF) {
            stonesoup_printf("%c", ch);
        }

        fclose(fifo);
    }
#if STONESOUP_TRACE
```

```
    tracepoint(stonesoup_trace, trace_point, "Finished reading sync file.");
#endif /* STONESOUP_TRACE */
}

void *calcDevamount(void *data) {
    struct stonesoup_data *stonesoupData = (struct stonesoup_data*)data;
    int qsize;

#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE,
"calcDevamount");
#endif /* STONESOUP_TRACE */

    stonesoup_printf("Inside calcDevAmount\n");

    pthread_mutex_lock(&stonesoup_mutex_0);

#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */

    /* STONESOUP: CROSSOVER-POINT (incorrect syncronization) */
    stonesoup_dev_amount = stonesoupData->data[0] - 'A';                       /*
oops...um... */
    qsize = stonesoupData->qsize;

    if (stonesoup_dev_amount < 0) {                                            /*
let's just clean up and */
        stonesoup_dev_amount *= -1;                                            /*
pretend that never happened */
    }

#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, variable_signed_integral, "stonesoup_dev_amount",
stonesoup_dev_amount, &stonesoup_dev_amount, "CROSSOVER-STATE");
#endif /* STONESOUP_TRACE */

    stonesoup_readFile(stonesoupData->file2);

    if (stonesoup_dev_amount == 0) {                                           /*
shhhh, just some more cleanup */
        stonesoup_dev_amount += 1;                                             /*
nothing to see here */
    }

#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "CROSSOVER-PONT: AFTER");
    tracepoint(stonesoup_trace, variable_signed_integral, "stonesoup_dev_amount",
stonesoup_dev_amount, &stonesoup_dev_amount, "FINAL-STATE");
#endif /* STONESOUP_TRACE */

    pthread_mutex_unlock(&stonesoup_mutex_0);
    return NULL;
}

void *devChar(void *data) {
    struct stonesoup_data *stonesoupData = (struct stonesoup_data*)data;
    int stonesoup_i;
    int i;
```

```
    int *stonesoup_arr = NULL;

#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE, "devChar");
#endif /* STONESOUP_TRACE */

    stonesoup_printf("Inside devChar\n");

    /* slow things down to make correct thing happen in good cases */
    stonesoup_arr = malloc(sizeof(int) * stonesoupData->qsize);

    pthread_mutex_lock(&stonesoup_mutex_1);
    for (stonesoup_i = 0; stonesoup_i < stonesoupData->qsize; stonesoup_i++) {
        stonesoup_arr[stonesoup_i] = stonesoupData->qsize - stonesoup_i;
    }
    qsort(stonesoup_arr, stonesoupData->qsize, sizeof(int), &stonesoup_comp);
    free(stonesoup_arr);

    stonesoup_readFile(stonesoupData->file1);

#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
    tracepoint(stonesoup_trace, variable_signed_integral, "stonesoup_dev_amount",
stonesoup_dev_amount, &stonesoup_dev_amount, "TRIGGER-STATE");
#endif /* STONESOUP_TRACE */

    /* STONESOUP: TRIGGER-POINT (incorrect syncronization) */
    for (i = 0; i < strlen(stonesoupData->data); i++) {                        /* can
cause underread/write if */
        stonesoupData->data[i] /= stonesoup_dev_amount;                        /*
stonesoup_dev_amount is neg */
    }

#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: AFTER");
#endif /* STONESOUP_TRACE */

    pthread_mutex_unlock(&stonesoup_mutex_1);
    return NULL;
}

#endif /* STONESOUP_SNIPPET_SUPPORT */

#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/******************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 ******************************************************************************/
void weakness(char * stonesoup_taintedValue) {
#define STONESOUP_TAINT_SOURCE stonesoup_taintedValue
#endif /* STONESOUP_SNIPPET_DEBUG */

#if STONESOUP_SNIPPET_BODY_DECLARATIONS
```

```
    struct stonesoup_data* stonesoupData;
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */

#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_start, "CWE821", "A", "Incorrect
Synchronization");
#endif /* STONESOUP_TRACE */

    stonesoupData =  malloc(sizeof(struct stonesoup_data));
    if (stonesoupData) {
        stonesoupData->data  = malloc(sizeof(char) * (strlen(STONESOUP_TAINT_SOURCE) +
1));
        stonesoupData->file1 = malloc(sizeof(char) * (strlen(STONESOUP_TAINT_SOURCE) +
1));
        stonesoupData->file2 = malloc(sizeof(char) * (strlen(STONESOUP_TAINT_SOURCE) +
1));
        if (stonesoupData->data) {
            if ((sscanf(STONESOUP_TAINT_SOURCE, "%d %s %s %s",
                      &(stonesoupData->qsize),
                        stonesoupData->file1,
                        stonesoupData->file2,
                        stonesoupData->data)  == 4) &&
                (strlen(stonesoupData->data)  != 0) &&
                (strlen(stonesoupData->file1) != 0) &&
                (strlen(stonesoupData->file2) != 0))
            {
                pthread_mutex_init(&stonesoup_mutex_0, NULL);
                pthread_mutex_init(&stonesoup_mutex_1, NULL);

#if STONESOUP_TRACE
                tracepoint(stonesoup_trace, variable_signed_integral,
"stonesoupData->qsize", stonesoupData->qsize, &(stonesoupData->qsize),
"INITIAL-STATE");
                tracepoint(stonesoup_trace, variable_buffer, "stonesoupData->data",
stonesoupData->data, "INITIAL-STATE");
                tracepoint(stonesoup_trace, variable_buffer, "stonesoupData->file1",
stonesoupData->file1, "INITIAL-STATE");
                tracepoint(stonesoup_trace, variable_buffer, "stonesoupData->file2",
stonesoupData->file2, "INITIAL-STATE");
                tracepoint(stonesoup_trace, trace_point, "Spawning threads.");
#endif /* STONESOUP_TRACE */

                if (strlen(stonesoupData->data) > 50) {
/* if size is large */

/*  iterate by different */
                    if (pthread_create(&stonesoup_t0, NULL, calcDevamount,
stonesoupData) != 0) {   /*  size (weakness). */
                        stonesoup_printf("Error initializing thread 0.");
                    }
                }

                if (pthread_create(&stonesoup_t1, NULL, devChar, stonesoupData) != 0)
{
                    stonesoup_printf("Error initializing thread 1.");
                }

                if (strlen(stonesoupData->data) > 50) {
```

```
                    pthread_join(stonesoup_t0, NULL);
                }
                pthread_join(stonesoup_t1, NULL);

#if STONESOUP_TRACE
                tracepoint(stonesoup_trace, trace_point, "Threads joined.");
#endif /* STONESOUP_TRACE */

                pthread_mutex_destroy(&stonesoup_mutex_0);
                pthread_mutex_destroy(&stonesoup_mutex_1);
            } else {
#if STONESOUP_TRACE
                tracepoint(stonesoup_trace, trace_error, "Error parsing data");
#endif /* STONESOUP_TRACE */
                stonesoup_printf("Error parsing data\n");
            }
            free(stonesoupData->data);
        }
        free(stonesoupData);
    }

#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */

#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 ******************************************************************************/
int main(int argc,char *argv[])
{
    char* tainted_buff;
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    tainted_buff = argv[1];

    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
```

```
    return 0;
}
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-822A Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1
#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1
#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1
#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1
#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1
#endif /* NOTHING DEFINED */

#if STONESOUP_SNIPPET_INCLUDES
/******************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 ******************************************************************************/
#include <stdlib.h> /* malloc */
#include <stdio.h> /* fprintf */
#include <string.h> /* memset */
#include <errno.h> /* errno */
/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_INCLUDES */
#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
/******************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 ******************************************************************************/
```

```
#include "output-handlers/handler.h"
#endif /* STONESOUP_SNIPPET_DEBUG */
typedef int (*stonesoup_fct_ptr)(const char *, const char *);
stonesoup_fct_ptr stonesoup_switch_func(char *param)
{
#if STONESOUP_TRACE
  tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE,
"stonesoup_switch_func");
#endif /* STONESOUP_TRACE */
  int var_len = 0;
  stonesoup_fct_ptr fct_ptr_addr = (stonesoup_fct_ptr )0;
  var_len = strlen(param) % 3;
  if (var_len == 0) {
    return strcmp;
  }
  else if (var_len == 1) {
    return strcoll;
  }
  else {
    sscanf(param,"%p",&fct_ptr_addr);
    return fct_ptr_addr;
  }
}
#endif /* STONESOUP_SNIPPET_SUPPORT */

#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 ******************************************************************************/
void weakness(char * stonesoup_tainted_buff) {
#define STONESOUP_TAINT_SOURCE stonesoup_tainted_buff
#endif /* STONESOUP_SNIPPET_DEBUG */
#if STONESOUP_SNIPPET_BODY_DECLARATIONS
 int stonesoup_cmp_flag = 0;
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_start, "CWE822", "A", "Untrusted Pointer
Dereference");
    tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
 /* STONESOUP: CROSSOVER-POINT (Untrusted Pointer Deference) */
 stonesoup_fct_ptr stonesoup_fp;
 const char *stonesoup_rand_word = "criticisms_metallide";
 stonesoup_fp = stonesoup_switch_func(STONESOUP_TAINT_SOURCE);
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
    tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
 /* STONESOUP: TRIGGER-POINT (Untrusted Pointer Dereference) */
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, variable_address, "stonesoup_fp", stonesoup_fp,
```

```
"TRIGGER-STATE");
#endif /* STONESOUP_TRACE */
    stonesoup_cmp_flag = ( *stonesoup_fp)(stonesoup_rand_word,STONESOUP_TAINT_SOURCE);
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
    if (stonesoup_cmp_flag == 0)
        stonesoup_printf("strings are equal\n");
    else
        stonesoup_printf("strings are not equal\n");
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 *****************************************************************************/
int main(int argc,char *argv[])
{
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    char* tainted_buff = argv[1];

    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
    return 0;
```

```
}
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-824A Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1
#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1
#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1
#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1
#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1
#endif /* NOTHING DEFINED */

#if STONESOUP_SNIPPET_INCLUDES
/******************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 ******************************************************************************/
#include <stdlib.h> /* malloc */
#include <string.h> /* memset */
#include <stdio.h> /* printf */
/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_INCLUDES */
#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
/******************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 ******************************************************************************/
#include "output-handlers/handler.h"
#endif /* STONESOUP_SNIPPET_DEBUG */
```

```
struct stonesoup_data_struct {
  int (*func_member)(char *);
  char *str_member;
};
int stonesoup_modulus_function(char *modulus_param_str)
{
#if STONESOUP_TRACE
  tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE,
"stonesoup_modulus_function");
#endif /* STONESOUP_TRACE */
  return modulus_param_str[0] % 2;
}
void stonesoup_set_function(char *set_param_str,struct stonesoup_data_struct
*set_param_data_struct)
{
#if STONESOUP_TRACE
  tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE,
"stonesoup_set_function");
  tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
/* STONESOUP: CROSSOVER-POINT (Uninitialized Pointer) */
  if (strlen(set_param_str) > 10U) {
    set_param_data_struct -> func_member = stonesoup_modulus_function;
    set_param_data_struct -> str_member = set_param_str;
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "Initialized pointer");
#endif /* STONESOUP_TRACE */
  }
  if (strlen(set_param_str) < 10U) {
    set_param_data_struct -> func_member = stonesoup_modulus_function;
    set_param_data_struct -> str_member = "default";
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "Initialized pointer");
#endif /* STONESOUP_TRACE */
  }
#if STONESOUP_TRACE
  tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
}
#endif /* STONESOUP_SNIPPET_SUPPORT */

#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 *****************************************************************************/
void weakness(char * stonesoup_tainted_buff) {
#define STONESOUP_TAINT_SOURCE stonesoup_tainted_buff
#endif /* STONESOUP_SNIPPET_DEBUG */
#if STONESOUP_SNIPPET_BODY_DECLARATIONS
    int stonesoup_val = 0;
    struct stonesoup_data_struct stonesoup_my_foo;
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */
```

```
#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_start, "CWE824", "A", "Access of
Uninitialized Pointer");
#endif /* STONESOUP_TRACE */
    if (strlen(STONESOUP_TAINT_SOURCE) < 1) {
        stonesoup_printf("string is too short to test\n");
    } else {
        stonesoup_set_function(STONESOUP_TAINT_SOURCE, &stonesoup_my_foo);
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
        /* STONESOUP: TRIGGER-POINT (Uninitialized Pointer) */
        stonesoup_val = (stonesoup_my_foo . func_member(stonesoup_my_foo .
str_member));
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
        if (stonesoup_val == 0)
            stonesoup_printf("mod is true\n");
        else
            stonesoup_printf("mod is false\n");
    }
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 ******************************************************************************/
int main(int argc,char *argv[])
{
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    char* tainted_buff = argv[1];

    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
    return 0;
```

```
}
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-824B Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1
#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1
#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1
#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1
#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1
#endif /* NOTHING DEFINED */

#if STONESOUP_SNIPPET_INCLUDES
/*****************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 *****************************************************************************/
#include <stdlib.h> /* malloc */
#include <stdio.h> /* fprintf */
#include <string.h> /* memset */
#include <errno.h> /* errno */
/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_INCLUDES */
#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 *****************************************************************************/
#include "output-handlers/handler.h"
```

```
#endif /* STONESOUP_SNIPPET_DEBUG */
typedef int (*fptr)();
int stonesoup_modulus_function1 (char *modulus_param_str) {
#if STONESOUP_TRACE
  tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE,
"stonesoup_modulus_function1");
#endif /* STONESOUP_TRACE */
  return modulus_param_str[0] % 2;
}
int stonesoup_modulus_function2 (char *modulus_param_str) {
#if STONESOUP_TRACE
  tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE,
"stonesoup_modulus_function2");
#endif /* STONESOUP_TRACE */
  return modulus_param_str[1] % 2;
}
void stonesoup_get_function(int len, fptr * modulus_function) {
#if STONESOUP_TRACE
  tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE,
"stonesoup_get_function");
    tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
    if (len > 10) {
        *modulus_function = stonesoup_modulus_function1;
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, trace_point, "Initialized pointer.");
#endif /* STONESOUP_TRACE */
    }
    if (len < 10) {
        *modulus_function = stonesoup_modulus_function2;
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, trace_point, "Initialized pointer.");
#endif /* STONESOUP_TRACE */
    }
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
}
#endif /* STONESOUP_SNIPPET_SUPPORT */

#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 *****************************************************************************/
void weakness(char * stonesoup_tainted_buff) {
#define STONESOUP_TAINT_SOURCE stonesoup_tainted_buff
#endif /* STONESOUP_SNIPPET_DEBUG */
#if STONESOUP_SNIPPET_BODY_DECLARATIONS
 int stonesoup_input_len = 0;
 int stonesoup_result = 0;
 fptr* stonesoup_function_ptr = 0;
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */
```

```
#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_start, "CWE824", "B", "Access of
Uninitialized Pointer");
#endif /* STONESOUP_TRACE */
 stonesoup_input_len = strlen(STONESOUP_TAINT_SOURCE);
    if (stonesoup_input_len < 2) {
        stonesoup_printf("String is too short to test\n");
    } else {
        stonesoup_function_ptr = malloc(sizeof(void *));
        if (stonesoup_function_ptr == 0) {
            stonesoup_printf("Error: Failed to allocate memory\n");
            exit(1);
        }
        /* STONESOUP: CROSSOVER-POINT (Uninitialized Pointer) */
        stonesoup_get_function(stonesoup_input_len, stonesoup_function_ptr);

#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
        /* STONESOUP: TRIGGER-POINT (Uninitialized Pointer) */
        stonesoup_result = ( *stonesoup_function_ptr)(STONESOUP_TAINT_SOURCE);
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
        if (stonesoup_result == 0)
            stonesoup_printf("mod is true\n");
        else
            stonesoup_printf("mod is false\n");

        if (stonesoup_function_ptr != 0) {
            free(stonesoup_function_ptr);
        }
    }
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 *****************************************************************************/
int main(int argc,char *argv[])
{
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    char* tainted_buff = argv[1];

    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
```

```
        stonesoup_close_printf_context();
    }
    return 0;
```

Approved for public release; distribution unlimited.

```
}
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-828A Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1

#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1

#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1

#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1

#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1

#endif /* NOTHING DEFINED */


#if STONESOUP_SNIPPET_INCLUDES
/******************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 ******************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <math.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>

/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */

#endif /* STONESOUP_SNIPPET_INCLUDES */
```

```
#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 ******************************************************************************/
#include "output-handlers/handler.h"

#endif /* STONESOUP_SNIPPET_DEBUG */

int SIZE = 50;
char *playful_platypus;

struct stonesoup_data {
    char *data;
};

struct stonesoup_data *stonesoupData;

int stonesoup_comp (const void * a, const void * b)
{
    if (a > b) {
        return -1;
    }
    else if (a < b) {
        return 1;
    }
    else {
        return 0;
    }
}

int stonesoup_pmoc (const void * a, const void * b)
{
    return -1 * stonesoup_comp(a, b);
}

void stonesoup_readFile(char *filename) {
    FILE *fifo;
    char ch;

#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE,
"stonesoup_readFile");
#endif /* STONESOUP_TRACE */

    fifo = fopen(filename, "r");

    if (fifo != NULL) {
        while ((ch = fgetc(fifo)) != EOF) {
            stonesoup_printf("%c", ch);
        }

        fclose(fifo);
```

```
    }
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "Finished reading sync file.");
#endif /* STONESOUP_TRACE */
}

void sig_handler (int sig) {
    stonesoup_printf("In sig_handler\n");

#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE,
"sig_handler");
    tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
    tracepoint(stonesoup_trace, variable_address, "playful_platypus",
playful_platypus, "TRIGGER-STATE");
#endif /* STONESOUP_TRACE */

    /* STONESOUP: TRIGGER-POINT (asyncunsafesighandler) */
    /* iterate through array and do something */
    if (playful_platypus[0] != '\0') {                      /* bad error checking -
can cause null ptr deref */
        stonesoup_printf(playful_platypus);
    }

#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE,
"TRIGGER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
}

void waitForSig(char *sleepFile) {
    int fd;
    char outStr[25] = {0};
    char filename[500] = {0};

#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE, "Wait for
sig.");
#endif /* STONESOUP_TRACE */

    stonesoup_printf("In waitForSig\n");

    sprintf(outStr, "%d.pid", getpid());
    strcat(filename, STONESOUP_TESTDATA);
    strcat(filename, outStr);

    if ((fd = open(filename, O_CREAT|O_WRONLY, 0666)) == -1) {
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, trace_error, "Error opening file.");
#endif /* STONESOUP_TRACE */
        stonesoup_printf("Error opening file.");
    }
    else {
        if (write(fd, "q", sizeof(char)) == -1) {
#if STONESOUP_TRACE
            tracepoint(stonesoup_trace, trace_error, "Error writing to file.");
#endif /* STONESOUP_TRACE */
            stonesoup_printf("Error writing to file.");
        }
```

```
        if (close(fd) == -1) {
#if STONESOUP_TRACE
            tracepoint(stonesoup_trace, trace_error, "Error closing file.");
#endif /* STONESOUP_TRACE */
            stonesoup_printf("Error closing file.");
        }

#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, trace_point, "Wrote .pid file");
#endif /* STONESOUP_TRACE */

        stonesoup_readFile(sleepFile);
    }
}

#endif /* STONESOUP_SNIPPET_SUPPORT */

#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/******************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 *****************************************************************************/
void weakness(char * stonesoup_taintedValue) {
#define STONESOUP_TAINT_SOURCE stonesoup_taintedValue
#endif /* STONESOUP_SNIPPET_DEBUG */

#if STONESOUP_SNIPPET_BODY_DECLARATIONS
    int stonesoup_i;
    char *temp;
    char *sleepFile;
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */

#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_start, "CWE828", "A", "Signal Handler with
Functionality that is not Asynchronous-safe.");
#endif /* STONESOUP_TRACE */

    stonesoupData =  malloc(sizeof(struct stonesoup_data));
    if (stonesoupData) {
        sleepFile = malloc(sizeof(char) * (strlen(STONESOUP_TAINT_SOURCE) + 1));
        stonesoupData->data = malloc(sizeof(char) * (strlen(STONESOUP_TAINT_SOURCE) +
1));
        if (stonesoupData->data) {
            if ((sscanf(STONESOUP_TAINT_SOURCE, "%s %s",
                        sleepFile,
                        stonesoupData->data) == 2) &&
                (strlen(stonesoupData->data) != 0) &&
                (strlen(sleepFile)           != 0))
            {

#if STONESOUP_TRACE
```

```
                tracepoint(stonesoup_trace, variable_buffer, "sleepFile", sleepFile,
"INITIAL_STATE");
                tracepoint(stonesoup_trace, variable_buffer, "stonesoupData->data",
stonesoupData->data, "INITIAL-STATE");
#endif /* STONESOUP_TRACE */

                if (signal(SIGUSR1, sig_handler) == SIG_ERR) {
#if STONESOUP_TRACE
                    tracepoint(stonesoup_trace, trace_error, "Error catching
SIGUSR1!");
#endif /* STONESOUP_TRACE */
                    stonesoup_printf ("Error catching SIGNUSR1!\n");
                }

                playful_platypus = malloc(sizeof(char) * (SIZE + 1));
                stonesoup_i = 0;
                while (stonesoupData->data[stonesoup_i] != '\0') {
/* copy input to global char* */
                    if (stonesoup_i < SIZE) {
                        playful_platypus[stonesoup_i] =
stonesoupData->data[stonesoup_i];
                        stonesoup_i++;
                    } else {                                              /* if input
size > 50 char, realloc size by hand */
                        playful_platypus[SIZE] = '\0';

#if STONESOUP_TRACE
                        tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT:
BEFORE");
#endif /* STONESOUP_TRACE */

                        /* STONESOUP: CROSSOVER-POINT (asyncunsafesighandler) */
                        SIZE *= 2;
                        temp = malloc(sizeof(char) * SIZE);
                        strcpy(temp, playful_platypus);
                        free(playful_platypus);
                        playful_platypus = NULL;                        /* calling sig
handler after this instruction to break */

#if STONESOUP_TRACE
                        tracepoint(stonesoup_trace, variable_address,
"playful_platypus", playful_platypus, "CROSSOVER-STATE");
#endif /* STONESOUP_TRACE */
                        waitForSig(sleepFile);
#if STONESOUP_TRACE
                        tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT:
AFTER");
#endif /* STONESOUP_TRACE */
                        playful_platypus = temp;
#if STONESOUP_TRACE
                        tracepoint(stonesoup_trace, variable_address,
"playful_platypus", playful_platypus, "FINAL-STATE");
#endif /* STONESOUP_TRACE */
                    }
                }
                free (playful_platypus);
                signal(SIGUSR1, SIG_IGN);           /* 'deregister' signal hander
befor returning to base program */
            } else {
```

```
#if STONESOUP_TRACE
                tracepoint(stonesoup_trace, trace_point, "Error parsing data");
#endif /* STONESOUP_TRACE */
                stonesoup_printf("Error parsing data\n");
            }
            free(stonesoupData->data);
        }
        free (stonesoupData);
    }
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */

#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 *******************************************************************************/
int main(int argc,char *argv[])
{
    char* tainted_buff;
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    tainted_buff = argv[1];

    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
```

```
    return 0;
}
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-831A Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1

#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1

#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1

#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1

#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1

#endif /* NOTHING DEFINED */


#if STONESOUP_SNIPPET_INCLUDES
/*****************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 *****************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <math.h>
#include <string.h>
#include <fcntl.h>

/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */

#endif /* STONESOUP_SNIPPET_INCLUDES */
```

```
#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 ******************************************************************************/
#include "output-handlers/handler.h"

#endif /* STONESOUP_SNIPPET_DEBUG */

struct stonesoup_data {
    char *data;
    char *file1;
    char *file2;
};

struct stonesoup_data *stonesoupData;

int stonesoup_loop;
int *stonesoup_global1;

int stonesoup_comp (const void * a, const void * b)
{
    if (a > b) {
        return -1;
    }
    else if (a < b) {
        return 1;
    }
    else {
        return 0;
    }
}

int stonesoup_pmoc (const void * a, const void * b)
{
    return -1 * stonesoup_comp(a, b);
}

void stonesoup_readFile(char *filename) {
    FILE *fifo;
    char ch;

#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE,
"stonesoup_readFile");
#endif /* STONESOUP_TRACE */

    fifo = fopen(filename, "r");

    if (fifo != NULL) {
        while ((ch = fgetc(fifo)) != EOF) {
            stonesoup_printf("%c", ch);
```

```
        }

        fclose(fifo);
    }

#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "Finished reading sync file.");
#endif /* STONESOUP_TRACE */
}

void waitForSig() {
    int fd;
    char outStr[25] = {0};
    char filename[500] = {0};

#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE,
"waitForSig");
#endif /* STONESOUP_TRACE */

    stonesoup_printf("In waitForSig\n");

    sprintf(outStr, "%d.pid", getpid());
    strcat(filename, STONESOUP_TESTDATA);
    strcat(filename, outStr);

    if ((fd = open(filename, O_CREAT|O_WRONLY, 0666)) == -1) {
#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, trace_error, "Error opening file.");
#endif /* STONESOUP_TRACE */
        stonesoup_printf("Error opening file.");
    }
    else {
        if (write(fd, "q", sizeof(char)) == -1) {
#if STONESOUP_TRACE
            tracepoint(stonesoup_trace, trace_error, "Error writing to file.");
#endif /* STONESOUP_TRACE */
            stonesoup_printf("Error writing to file.");
        }

        if (close(fd) == -1) {
#if STONESOUP_TRACE
            tracepoint(stonesoup_trace, trace_error, "Error closing file.");
#endif /* STONESOUP_TRACE */
            stonesoup_printf("Error closing file.");
        }

#if STONESOUP_TRACE
        tracepoint(stonesoup_trace, trace_point, "Finished writing .pid file.");
#endif /* STONESOUP_TRACE */

        stonesoup_printf("Reading file1\n");
        stonesoup_readFile(stonesoupData->file1);
        stonesoup_readFile(stonesoupData->file2);
    }
}

void stonesoup_sig_handler (int sig) {
    stonesoup_printf("In stonesoup_sig_handler\n");
```

```
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE,
"stonesoup_sig_handler");
    tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
    tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */

    /* STONESOUP: CROSSOVER-POINT (signal handler for multiple signals) */
    /* STONESOUP: TRIGGER-POINT (signal handler for multiple signals) */
    stonesoup_global1[0] = -1;
    free(stonesoup_global1);
    stonesoup_global1 = NULL;

#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: AFTER");
    tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
#endif /* STONESOUP_TRACE */

    stonesoup_printf("In sig handler");
}

#endif /* STONESOUP_SNIPPET_SUPPORT */

#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 *****************************************************************************/
void weakness(char * stonesoup_taintedValue) {
#define STONESOUP_TAINT_SOURCE stonesoup_taintedValue
#endif /* STONESOUP_SNIPPET_DEBUG */

#if STONESOUP_SNIPPET_BODY_DECLARATIONS
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */

#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_start, "CWE831", "A", "Signal Handler
Function Associated with Multiple Signals");
#endif /* STONESOUP_TRACE */

    stonesoupData =  malloc(sizeof(struct stonesoup_data));
    if (stonesoupData) {
        stonesoupData->data  = malloc(sizeof(char) * (strlen(STONESOUP_TAINT_SOURCE) +
1));
        stonesoupData->file1 = malloc(sizeof(char) * (strlen(STONESOUP_TAINT_SOURCE) +
1));
        stonesoupData->file2 = malloc(sizeof(char) * (strlen(STONESOUP_TAINT_SOURCE) +
1));
        if (stonesoupData->data) {
            if ((sscanf(STONESOUP_TAINT_SOURCE, "%s %s %s",
                        stonesoupData->file1,
```

```
                        stonesoupData->file2,
                        stonesoupData->data)  == 3) &&
                (strlen(stonesoupData->data)  != 0) &&
                (strlen(stonesoupData->file1) != 0) &&
                (strlen(stonesoupData->file2) != 0))
            {
                stonesoup_global1 = calloc(1, sizeof(int));

#if STONESOUP_TRACE
                tracepoint(stonesoup_trace, variable_buffer, "stonesoupData->data",
stonesoupData->data, "INITIAL-STATE");
                tracepoint(stonesoup_trace, variable_buffer, "stonesoupData->file1",
stonesoupData->file1, "INITIAL-STATE");
                tracepoint(stonesoup_trace, variable_buffer, "stonesoupData->file2",
stonesoupData->file2, "INITIAL-STATE");
#endif /* STONESOUP_TRACE */

                /* optionally set up sig handler bassed on input */
                if (signal(SIGUSR1, stonesoup_sig_handler) == SIG_ERR) {
#if STONESOUP_TRACE
                    tracepoint(stonesoup_trace, trace_error, "Error catching
SIGUSR1");
#endif /* STONESOUP_TRACE */
                    stonesoup_printf ("Error catching SIGUSR1!\n");
                }
                stonesoup_printf("Set up SIGUSR1 handler\n");

                if (stonesoupData->data[0] >= 'A' && stonesoupData->data[0] <= 'Z') {
                    if (signal(SIGUSR2, stonesoup_sig_handler) == SIG_ERR) {
#if STONESOUP_TRACE
                        tracepoint(stonesoup_trace, trace_error, "Error catching
SIGUSR2");
#endif /* STONESOUP_TRACE */
                        stonesoup_printf ("Error catching SIGUSR2!\n");
                    }

                    stonesoup_printf("Set up SIGUSR2 handler\n");
                }
                waitForSig();
                stonesoup_printf("After waitForSig\n");
                signal(SIGUSR1, SIG_IGN);
/* "deregister" sig handler */
                signal(SIGUSR2, SIG_IGN);
/*   before moving on */
                if (stonesoup_global1 != NULL) {
                    free(stonesoup_global1);
                    stonesoup_global1 = NULL;
                }
            } else {
#if STONESOUP_TRACE
                tracepoint(stonesoup_trace, trace_error, "Error parsing data.");
#endif /* STONESOUP_TRACE */
                stonesoup_printf("Error parsing data\n");
            }
            free(stonesoupData->data);
        }
        free (stonesoupData);
    }
#if STONESOUP_TRACE
```

```
    tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */

#if STONESOUP_SNIPPET_DEBUG
/******************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 ******************************************************************************/
int main(int argc,char *argv[])
{
    char* tainted_buff;
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    tainted_buff = argv[1];

    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
```

```
    return 0;
}
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-833A Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1

#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1

#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1

#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1

#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1

#endif /* NOTHING DEFINED */


#if STONESOUP_SNIPPET_INCLUDES
/*******************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 ******************************************************************************/
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <string.h>

/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */

#endif /* STONESOUP_SNIPPET_INCLUDES */

#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
```

```
/*******************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 ******************************************************************************/
#include "output-handlers/handler.h"

#endif /* STONESOUP_SNIPPET_DEBUG */

struct stonesoup_data {
    int qsize;
    char *data;
    char *file1;
    char *file2;
};

pthread_mutex_t stonesoup_mutex_0, stonesoup_mutex_1;
pthread_t stonesoup_t0, stonesoup_t1;

int stonesoup_comp (const void * a, const void * b)
{
    if (a > b) {
        return -1;
    }
    else if (a < b) {
        return 1;
    }
    else {
        return 0;
    }
}

int stonesoup_pmoc (const void * a, const void * b)
{
    return -1 * stonesoup_comp(a, b);
}

void stonesoup_readFile(char *filename) {
    FILE *fifo;
    char ch;

#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE,
"stonesoup_readFile");
#endif /* STONESOUP_TRACE */

    fifo = fopen(filename, "r");

    if (fifo != NULL) {
        while ((ch = fgetc(fifo)) != EOF) {
            stonesoup_printf("%c", ch);
        }

        fclose(fifo);
    }
```

```
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "Finished reading sync file.");
#endif /* STONESOUP_TRACE */
}

void *stonesoup_replace (void *data) {
    struct stonesoup_data *stonesoupData = (struct stonesoup_data*)data;
    int *qsort_arr;
    int i = 0;

#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE,
"stonesoup_replace");
    tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */

    stonesoup_printf("replace: entering function\n");

    /* slow things down to make correct thing happen in good cases */
    qsort_arr = malloc(sizeof(int)*stonesoupData->qsize);
    if (qsort_arr != NULL) {
        for (i = 0; i < stonesoupData->qsize; i++) {
            qsort_arr[i] = stonesoupData->qsize - i;
        }
        qsort(qsort_arr, stonesoupData->qsize, sizeof(int), &stonesoup_comp);
        free (qsort_arr);
        qsort_arr = NULL;
    }

    stonesoup_readFile(stonesoupData->file1);

    stonesoup_printf("replace: Attempting to grab lock 0\n");
    pthread_mutex_lock(&stonesoup_mutex_0);
    stonesoup_printf("replace: Grabbed lock 0\n");


    stonesoup_printf("replace: Attempting to grab lock 1\n");
    pthread_mutex_lock(&stonesoup_mutex_1);                              /* DEADLOCK */
    stonesoup_printf("replace: Grabbed lock 1\n");

    i = 0;
    while(stonesoupData->data[i] != '\0') {
        if (stonesoupData->data[i] == '_') {
            stonesoupData->data[i] = '-';
        }
        i++;
    }

    stonesoup_printf("replace: Releasing lock 1\n");
    pthread_mutex_unlock(&stonesoup_mutex_1);
    stonesoup_printf("replace: Releasing lock 0\n");
    pthread_mutex_unlock(&stonesoup_mutex_0);

#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
    return NULL;
}
```

```
void *stonesoup_toCap (void *data) {
    struct stonesoup_data *stonesoupData = (struct stonesoup_data*)data;
    int i = 0;

#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE,
"stonesoup_toCap");
#endif /* STONESOUP_TRACE */

    stonesoup_printf("toCap:   Entering function\n");
    stonesoup_printf("toCap:   Attempting to grab lock 1\n");
    pthread_mutex_lock(&stonesoup_mutex_1);
    stonesoup_printf("toCap:   Grabbed lock 1\n");

    stonesoup_readFile(stonesoupData->file2);

#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
    /* STONESOUP: TRIGGER-POINT (deadlock) */
    stonesoup_printf("toCap:   Attempting to grab lock 0\n");
    pthread_mutex_lock(&stonesoup_mutex_0);                             /* DEADLOCK */
    stonesoup_printf("toCap:   Grabbed lock 0\n");
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: AFTER");
#endif /* STONESOUP_TRACE */

    i = 0;
    while(stonesoupData->data[i] != '\0') {
        if (stonesoupData->data[i] > 'a' && stonesoupData->data[i] < 'z') {
            stonesoupData->data[i] -= 'a' - 'A';
        }
        i++;
    }

    stonesoup_printf("toCap:   Releasing lock 0\n");
    pthread_mutex_unlock(&stonesoup_mutex_0);
    stonesoup_printf("toCap:   Releasing lock 1\n");
    pthread_mutex_unlock(&stonesoup_mutex_1);
    return NULL;
}

#endif /* STONESOUP_SNIPPET_SUPPORT */

#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 *****************************************************************************/
void weakness(char * stonesoup_taintedValue) {
#define STONESOUP_TAINT_SOURCE stonesoup_taintedValue
#endif /* STONESOUP_SNIPPET_DEBUG */
```

```
#if STONESOUP_SNIPPET_BODY_DECLARATIONS
    int stonesoup_hasUnderscores = 0;
    int stonesoup_i = 0;
    struct stonesoup_data* stonesoupData;
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */

#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_start, "CWE833", "A", "Deadlock");
#endif /* STONESOUP_TRACE */

    stonesoupData =  malloc(sizeof(struct stonesoup_data));
    if (stonesoupData) {
        stonesoupData->data  = malloc(sizeof(char) * (strlen(STONESOUP_TAINT_SOURCE) +
1));
        stonesoupData->file1 = malloc(sizeof(char) * (strlen(STONESOUP_TAINT_SOURCE) +
1));
        stonesoupData->file2 = malloc(sizeof(char) * (strlen(STONESOUP_TAINT_SOURCE) +
1));
        if (stonesoupData->data) {
            if ((sscanf(STONESOUP_TAINT_SOURCE, "%d %s %s %s",
                      &(stonesoupData->qsize),
                        stonesoupData->file1,
                        stonesoupData->file2,
                        stonesoupData->data)  == 4) &&
                (strlen(stonesoupData->data)  != 0) &&
                (strlen(stonesoupData->file1) != 0) &&
                (strlen(stonesoupData->file2) != 0))
            {
#if STONESOUP_TRACE
                tracepoint(stonesoup_trace, variable_signed_integral,
"stonesoupData->qsize", stonesoupData->qsize, &(stonesoupData->qsize),
"INITIAL-STATE");
                tracepoint(stonesoup_trace, variable_buffer, "stonesoupData->data",
stonesoupData->data, "INITIAL-STATE");
                tracepoint(stonesoup_trace, variable_buffer, "stonesoupData->file1",
stonesoupData->file1, "INITIAL-STATE");
                tracepoint(stonesoup_trace, variable_buffer, "stonesoupData->file2",
stonesoupData->file2, "INITIAL-STATE");
#endif /* STONESOUP_TRACE */

                pthread_mutex_init(&stonesoup_mutex_0, NULL);
                pthread_mutex_init(&stonesoup_mutex_1, NULL);
                while(stonesoupData->data[stonesoup_i] != '\0') {                   /*
if the input contains underscores */
                    if (stonesoupData->data[stonesoup_i++] == '_') {                /*
we call the deadlocking function */
                        stonesoup_hasUnderscores = 1;
                    }
                }
#if STONESOUP_TRACE
                tracepoint(stonesoup_trace, trace_point, "Spawning threads.");
#endif /* STONESOUP_TRACE */
                if (pthread_create(&stonesoup_t0, NULL, stonesoup_toCap,
stonesoupData) != 0) {
                    stonesoup_printf("Thread 0 failed to spawn.");
                }
                if (stonesoup_hasUnderscores == 1) {
                    /* STONESOUP: CROSSOVER-POINT (deadlock) */
```

```
                    if (pthread_create(&stonesoup_t1, NULL, stonesoup_replace,
stonesoupData) != 0) {
                        stonesoup_printf("Thread 1 failed to spawn.");
                    }
                }

                pthread_join(stonesoup_t0, NULL);
                if (stonesoup_hasUnderscores == 1) {
                    pthread_join(stonesoup_t1, NULL);
                }
#if STONESOUP_TRACE
                tracepoint(stonesoup_trace, trace_point, "Threads joined.");
#endif /* STONESOUP_TRACE */
                pthread_mutex_destroy(&stonesoup_mutex_0);
                pthread_mutex_destroy(&stonesoup_mutex_1);
            } else {
#if STONESOUP_TRACE
                tracepoint(stonesoup_trace, trace_error, "Error parsing data");
#endif /* STONESOUP_TRACE */
                stonesoup_printf("Error parsing data\n");
            }
            free(stonesoupData->data);
        }
        free(stonesoupData);
    }
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */

#if STONESOUP_SNIPPET_DEBUG
/******************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 ******************************************************************************/
int main(int argc,char *argv[])
{
    char* tainted_buff;
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    tainted_buff = argv[1];

    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
```

```
    return 0;
}
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-839A Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1
#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1
#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1
#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1
#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1
#endif /* NOTHING DEFINED */

#if STONESOUP_SNIPPET_INCLUDES
/*****************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 *****************************************************************************/
#include <stdlib.h> // exit
#include <stdio.h> // vfprintf
#include <string.h> // memset
/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_INCLUDES */
#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 *****************************************************************************/
#include "output-handlers/handler.h"
```

```
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_SUPPORT */
#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 *****************************************************************************/
void weakness(char * stonesoup_tainted_buff) {
#define STONESOUP_TAINT_SOURCE stonesoup_tainted_buff
#endif /* STONESOUP_SNIPPET_DEBUG */
#if STONESOUP_SNIPPET_BODY_DECLARATIONS
    char *stonesoup_buffer = 0U;
    int stonesoup_len;
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_start, "CWE839", "A", "Numeric Range
Comparison Without Minimum Check");
#endif /* STONESOUP_TRACE */
    stonesoup_len = atoi(STONESOUP_TAINT_SOURCE);
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
/* STONESOUP: CROSSOVER-POINT (No Minimum Check) */
    if (stonesoup_len < 4096) {
        stonesoup_buffer = ((char *)(malloc(4096 * sizeof(char ))));
        if (stonesoup_buffer != 0) {
            memset(stonesoup_buffer,'a',4096);
#if STONESOUP_TRACE
            tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
            tracepoint(stonesoup_trace, variable_signed_integral, "stonesoup_len",
stonesoup_len, &stonesoup_len, "TRIGGER-STATE");
#endif /* STONESOUP_TRACE */
/* STONESOUP: TRIGGER-POINT (No Minimum Check) */
            memset(&stonesoup_buffer[stonesoup_len],'b',4096 - stonesoup_len);
#if STONESOUP_TRACE
            tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: AFTER");
#endif /* STONESOUP_TRACE */
            stonesoup_buffer[4095] = 0;
            stonesoup_printf("%s\n",stonesoup_buffer);
            free(stonesoup_buffer);
        }
    }  else {
        stonesoup_printf("Number is too large to use\n");
    }
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
    tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
```

```
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 *****************************************************************************/
int main(int argc,char *argv[])
{
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    char* tainted_buff = argv[1];

    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
    exit(0);
```

```
}
#endif /* STONESOUP_SNIPPET_DEBUG */
```

C - CWE-843A Source Code

```
#if STONESOUP_SNIPPET_DEBUG || \
        (!STONESOUP_SNIPPET_INCLUDES && \
            !STONESOUP_SNIPPET_SUPPORT && \
            !STONESOUP_SNIPPET_BODY_STATEMENTS && \
            !STONESOUP_SNIPPET_BODY_DECLARATIONS)
/* Default to debug mode if nothing requested */
#define STONESOUP_SNIPPET_DEBUG 1
#undef STONESOUP_SNIPPET_INCLUDES
#define STONESOUP_SNIPPET_INCLUDES 1
#undef STONESOUP_SNIPPET_SUPPORT
#define STONESOUP_SNIPPET_SUPPORT 1
#undef STONESOUP_SNIPPET_BODY_STATEMENTS
#define STONESOUP_SNIPPET_BODY_STATEMENTS 1
#undef STONESOUP_SNIPPET_BODY_DECLARATIONS
#define STONESOUP_SNIPPET_BODY_DECLARATIONS 1
#endif /* NOTHING DEFINED */

#if STONESOUP_SNIPPET_INCLUDES
/*****************************************************************************
 * Weakness Snippet Dependencies
 *
 * This section should #include any headers for required functions, types,
 * macros, etc. in either the weakness or the supporting functions.
 *****************************************************************************/
#include <stdlib.h> /* malloc */
#include <string.h> /* memset */
#include <stdio.h> /* printf */
/*
* Include tracepoint events.
*/
#if STONESOUP_TRACE
#if STONESOUP_SNIPPET_DEBUG
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#endif /* STONESOUP_SNIPPET_DEBUG */
#include <stonesoup/stonesoup_trace.h>
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_INCLUDES */
#if STONESOUP_SNIPPET_SUPPORT
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet Support Functions
 *
 * This section should include any additional methods to support the defined
 * weakness snippet in the weakness() function.  If any functions should
 * not appear in the injected base program, simply wrap them in the
 * STONESOUP_SNIPPET_DEBUG macro, and they will be stripped by the pre-
 * processor.
 *****************************************************************************/
#include "output-handlers/handler.h"
```

```
#endif /* STONESOUP_SNIPPET_DEBUG */
struct stonesoup_message_buffer {
  union {
    int name_id_member;
    char *name_member;
  } message_data;
  int message_type;
};
int stonesoup_process_buf(void *param)
{
#if STONESOUP_TRACE
  tracepoint(stonesoup_trace, trace_location, STONESOUP_INJECTED_FILE,
"stonesoup_process_buf");
#endif /* STONESOUP_TRACE */
  struct stonesoup_message_buffer *message_param = 0;
  message_param = ((struct stonesoup_message_buffer *)param);
  if (message_param -> message_type == 1)
    return strlen(message_param -> message_data . name_member);
  else
    return message_param -> message_data . name_id_member;
}
#endif /* STONESOUP_SNIPPET_SUPPORT */

#if STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS
#if STONESOUP_SNIPPET_DEBUG
/*****************************************************************************
 * Weakness Snippet
 *
 * This section should include the weakness snippet which will be sourced in
 * to the injection point after all code complexities have been applied.  The
 * taint source is mapped to the weakness snippet using the macro:
 * STONESOUP_TAINT_SOURCE.  In debug mode, the function declaration is also
 * included.
 *****************************************************************************/
void weakness(char * stonesoup_tainted_buff) {
#define STONESOUP_TAINT_SOURCE stonesoup_tainted_buff
#endif /* STONESOUP_SNIPPET_DEBUG */
#if STONESOUP_SNIPPET_BODY_DECLARATIONS
    int stonesoup_val;
    unsigned long stonesoup_id;
    struct stonesoup_message_buffer stonesoup_buf;
#endif /* STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_BODY_STATEMENTS
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, weakness_start, "CWE843", "A", "Access of Resource
Using Incompatible Type");
#endif /* STONESOUP_TRACE */
    stonesoup_buf . message_type = 1;
    stonesoup_buf . message_data . name_member = STONESOUP_TAINT_SOURCE;
    stonesoup_id = atoi(STONESOUP_TAINT_SOURCE);
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, variable_signed_integral,
"stonesoup_buf.message_type", stonesoup_buf.message_type, &stonesoup_buf.message_type,
"INITIAL-STATE");
    tracepoint(stonesoup_trace, variable_buffer,
"stonesoup_buf.message_data.name_member", stonesoup_buf.message_data.name_member,
"INITIAL-STATE");
    tracepoint(stonesoup_trace, variable_signed_integral, "stonesoup_id",
stonesoup_id, &stonesoup_id, "INITIAL-STATE");
```

```
    tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
/* STONESOUP: CROSSOVER-POINT (Access From Incompatible Type) */
    if (stonesoup_id != 0)
        stonesoup_buf . message_data . name_id_member = stonesoup_id;
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, variable_signed_integral,
"stonesoup_buf.message_data.name_id_member",
stonesoup_buf.message_data.name_id_member, &stonesoup_buf.message_data.name_id_member,
"CROSSOVER-STATE");
    tracepoint(stonesoup_trace, trace_point, "CROSSOVER-POINT: AFTER");
    tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: BEFORE");
#endif /* STONESOUP_TRACE */
/* STONESOUP: TRIGGER-POINT (Access From Incompatible Type) */
    stonesoup_val = stonesoup_process_buf(&stonesoup_buf);
    stonesoup_printf("processing result is %i\n", stonesoup_val);
#if STONESOUP_TRACE
    tracepoint(stonesoup_trace, trace_point, "TRIGGER-POINT: AFTER");
    tracepoint(stonesoup_trace, variable_signed_integral, "stonesoup_val",
stonesoup_val, &stonesoup_val, "FINAL-STATE");
    tracepoint(stonesoup_trace, weakness_end);
#endif /* STONESOUP_TRACE */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS */
#if STONESOUP_SNIPPET_DEBUG
}
#endif /* STONESOUP_SNIPPET_DEBUG */
#endif /* STONESOUP_SNIPPET_BODY_STATEMENTS || STONESOUP_SNIPPET_BODY_DECLARATIONS */
#if STONESOUP_SNIPPET_DEBUG
/*******************************************************************************
 * Main
 *
 * This only exists to support direct debugging.
 *******************************************************************************/
int main(int argc,char *argv[])
{
    if (argc < 2) {
        printf("Error: requires a single command-line argument\n");
        exit(1);
    }
    char* tainted_buff = argv[1];
    if (tainted_buff != NULL) {
        stonesoup_setup_printf_context();
        weakness(tainted_buff);
        stonesoup_close_printf_context();
    }
    return 0;
```

```
}
#endif /* STONESOUP_SNIPPET_DEBUG */
```

Java Weakness Source Code

A collection of the source code for each Java weakness algorithmic variant.

Concurrency Handling Source

- J - CWE-363A Source Code
- J - CWE-367A Source Code
- J - CWE-412A Source Code
- J - CWE-414A Source Code
- J - CWE-543A Source Code
- J - CWE-567A Source Code
- J - CWE-572A Source Code
- J - CWE-609A Source Code
- J - CWE-663A Source Code
- J - CWE-764A Source Code
- J - CWE-765A Source Code
- J - CWE-820A Source Code
- J - CWE-821A Source Code
- J - CWE-832A Source Code
- J - CWE-833A Source Code

Error Handling Source

- J - CWE-209A Source Code
- J - CWE-248A Source Code
- J - CWE-252A Source Code
- J - CWE-252B Source Code
- J - CWE-253A Source Code
- J - CWE-390A Source Code
- J - CWE-391A Source Code
- J - CWE-460A Source Code
- J - CWE-584A Source Code

Injection Source

Content by label

There is no content with the specified labels

Number Handling Source

- J - CWE-190A Source Code
- J - CWE-190B Source Code
- J - CWE-191A Source Code
- J - CWE-194A Source Code
- J - CWE-195A Source Code
- J - CWE-196A Source Code
- J - CWE-197A Source Code
- J - CWE-369A Source Code
- J - CWE-839A Source Code

Resource Drains Source

Content by label

There is no content with the specified labels

Tainted Data Source

Content by label

There is no content with the specified labels

J - CWE-190A Source Code

```
package com.pontetec.weakness.numberhandling;

import java.io.PrintStream;
import com.pontetec.stonesoup.trace.Tracer;

public class CWE190A {
```

```
    public static void weakness(final short smallValue,
            PrintStream output){
        /*
         * CWE-190 Integer Overflow or Wraparound
         * This test takes in a value from a taint source,
         * and creates a string array based on the value.
         * Math is performed on the value such that, if the value is
         * greater than 500, the resultant value 'wraps around', and becomes
         * negative.  This results in an exception being thrown, due to the
         * invalid array size.
         */
        // Good values:  Any number less than 500
        // Bad values:    Any number greater than or equal to 500
        Tracer.tracepointWeaknessStart("CWE190", "A", "Integer Overflow or
Wraparound");

        short stonesoup_checked_value = smallValue;

        Tracer.tracepointVariableShort("stonesoup_checked_value",
stonesoup_checked_value);

        if (stonesoup_checked_value < 0) {
            stonesoup_checked_value = 0;
        }

        Tracer.tracepointVariableShort("stonesoup_checked_value",
stonesoup_checked_value);

        Tracer.tracepointMessage("CROSSOVER-POINT: BEFORE");
        /* STONESOUP: crossover point (user controlled value) */
        short stonesoup_value = (short) (((short)(Short.MAX_VALUE) -
                        (short)500) + stonesoup_checked_value);

        Tracer.tracepointVariableShort("stonesoup_value", stonesoup_value);

        Tracer.tracepointMessage("CROSSOVER-POINT: AFTER");

        stonesoup_value++;
        String[] stonesoup_array = null;
        try {
            Tracer.tracepointMessage("TRIGGER-POINT: BEFORE");
            /* STONESOUP: trigger point (NegativeArraySizeException thrown) */
            Tracer.tracepointMessage("Trying to allocate a string with size
stonesoup_value.");
            stonesoup_array = new String[stonesoup_value];
            Tracer.tracepointBufferInfo("stonesoup_array", stonesoup_array.length,
"Length of newly allocated stonesoup_array");
            Tracer.tracepointMessage("TRIGGER-POINT: AFTER");

            for (short index = 0; index < stonesoup_value; index++) {
                stonesoup_array[index] = Character.toString((char)index);
            }
            Tracer.tracepointMessage("Copy data into stonesoup_array.");
        } catch (java.lang.RuntimeException e) {
            Tracer.tracepointError(e.getClass().getName() + ": " + e.getMessage());
            e.printStackTrace(output);
            throw e;
        }
```

```
        for (int counter = 0; counter < stonesoup_array.length; counter++){
            output.printf("array[%d]=%s\n", counter, stonesoup_array[counter]);
        }

        Tracer.tracepointWeaknessEnd();
    }

    /**
     * @param args
     */
    public static void main(String[] args){
        short value = 0;
        String sourceTaint = args[0];
        if (sourceTaint == null) {
            throw new IllegalArgumentException("No source taint provided in
environment variable args[0].");
        }
        try {
            value = Short.parseShort(sourceTaint);
        } catch (NumberFormatException e) {
            throw new IllegalArgumentException("Source taint is not a valid short.",
e);
        }
        weakness(value, System.err);
    }
```

```
}
```

J - CWE-190B Source Code

```
package com.pontetec.weakness.numberhandling;

import java.io.PrintStream;
import com.pontetec.stonesoup.trace.Tracer;

public class CWE190B {

    public static void weakness(final short value,
            PrintStream output){
         /*
         * CWE-190 Integer Overflow or Wraparound
         * This test takes in a value from a taint source.
         * Math is performed on the value such that, if the
         * value is 32767 or 32766, the resultant value 'wraps around',
         * and becomes negative.  This results in an infinite loop.
         */
        // Good values: -32768 .. 32765
        // Bad values: 32767, 32766 */
        Tracer.tracepointWeaknessStart("CWE190", "B", "Integer Overflow or
Wraparound");

        short stonesoup_checked_value = value;

        Tracer.tracepointVariableShort("stonesoup_checked_value",
stonesoup_checked_value);

        if (stonesoup_checked_value <= 0) {
            stonesoup_checked_value = 1;
            output.println("resetting value to 1");
        }

        Tracer.tracepointVariableShort("stonesoup_checked_value",
stonesoup_checked_value);

        short stonesoup_counter = 2;

        Tracer.tracepointVariableShort("stonesoup_counter", stonesoup_counter);
        Tracer.tracepointMessage("CROSSOVER-POINT: BEFORE");

        /* Counter used for lttng logging, only log once every 20 loop iterations do
keep logging size down  */
        int lttngCtr = 99;

        /* STONESOUP: crossover point (user controlled value) */
        while(stonesoup_counter < 10){
            output.println("Loop #" + stonesoup_counter);
            if (stonesoup_counter > 0){
                Tracer.tracepointMessage("TRIGGER-POINT: BEFORE");
                /* STONESOUP: trigger point (overflows, causing infinite loop) */
                stonesoup_counter += stonesoup_checked_value;
            }
```

```
            if (stonesoup_counter > 0 || ++lttngCtr >= 100) {
                lttngCtr = 1;
                Tracer.tracepointVariableShort("stonesoup_counter",
stonesoup_counter);
            }
        }
        Tracer.tracepointMessage("TRIGGER-POINT: AFTER");
        Tracer.tracepointMessage("CROSSOVER-POINT: AFTER");
        Tracer.tracepointVariableShort("stonesoup_counter", stonesoup_counter);
        output.println("finished evaluating");

        Tracer.tracepointWeaknessEnd();
    }

    /**
     * @param args
     */
    public static void main(String[] args){
        short value = 0;
        String sourceTaint = args[0];
        if (sourceTaint == null) {
            throw new IllegalArgumentException("No source taint provided in
environment variable args[0].");
        }
        try {
            value = Short.parseShort(sourceTaint);
        } catch (NumberFormatException e) {
            throw new IllegalArgumentException("Source taint is not a valid short.",
e);
        }
        weakness(value, System.err);
    }
```

```
}
```

J - CWE-191A Source Code

```
package com.pontetec.weakness.numberhandling;

import java.io.PrintStream;
import com.pontetec.stonesoup.trace.Tracer;

public class CWE191A {

    public static void weakness(final short value, PrintStream output) {
        /*
         * CWE-191 Integer Underflow (Wrap or Wraparound) This test takes in a
         * value from a taint source, and creates a string array based on the
         * value. Math is performed on the value such that, if the value is
         * 32749 or greater, the resultant value underflows and becomes
         * positive. This results in an infinite loop.
         */
        // Good value: Positive value between 1 and 32748, inclusive
        // Bad value: 32749 - 32767

        Tracer.tracepointWeaknessStart("CWE191", "A", "Integer Underflow (Wrap or
Wraparound)");

        short stonesoup_checked_value = value;

        Tracer.tracepointVariableShort("stonesoup_checked_value",
stonesoup_checked_value);

        if (stonesoup_checked_value < 0) {
            stonesoup_checked_value = 0;
        }

        Tracer.tracepointVariableShort("stonesoup_checked_value",
stonesoup_checked_value);

        Short[] stonesoup_some_values = new Short[] { 0, 1, 2, 3, 4, 5, 6, 7,
                8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20 };
        short stonesoup_counter = -20;
        short stonesoup_offset = 40;

        Tracer.tracepointBufferInfo("stonesoup_some_values",
stonesoup_some_values.length, "Length of stonesoup_some_values");
        Tracer.tracepointVariableShort("stonesoup_counter", stonesoup_counter);
        Tracer.tracepointVariableShort("stonesoup_offset", stonesoup_offset);

        /* counter to allow lttng logging to occure every 100 iterations of the loop
*/
        int lttngCtr = 99;
        Tracer.tracepointMessage("CROSSOVER-POINT: BEFORE");
        Tracer.tracepointMessage("TRIGGER-POINT: BEFORE");
        /* STONESOUP: crossover point (user controlled value) */
        while ((stonesoup_counter + stonesoup_offset > 0)
                && (stonesoup_counter + stonesoup_offset <
```

```
stonesoup_some_values.length)) {
            output.printf("stonesoup_some_values[%d] : %s\n", stonesoup_counter
                    + stonesoup_offset, stonesoup_some_values[stonesoup_counter
                    + stonesoup_offset]);

            if (++lttngCtr >= 100) {
                Tracer.tracepointVariableShort("stonesoup_counter",
stonesoup_counter);
            }

            /* STONESOUP: trigger point (underflows, causing infinite loop) */
            stonesoup_counter -= stonesoup_checked_value;
            if (stonesoup_counter > -20) {
                stonesoup_counter = -20;
            }

            if (lttngCtr >= 100) {
                lttngCtr = 1;
                Tracer.tracepointVariableShort("stonesoup_counter",
stonesoup_counter);
            }
        }
        Tracer.tracepointMessage("TRIGGER-POINT: AFTER");
        Tracer.tracepointMessage("CROSSOVER-POINT: AFTER");
        Tracer.tracepointBufferInfo("stonesoup_some_values",
stonesoup_some_values.length, "Length of stonesoup_some_values");
        Tracer.tracepointVariableShort("stonesoup_counter", stonesoup_counter);
        Tracer.tracepointVariableShort("stonesoup_offset", stonesoup_offset);
        output.println("finished evaluating");

        Tracer.tracepointWeaknessEnd();
    }

    /**
     * @param args
     */
    public static void main(String[] args) {
        short value = 0;
        String sourceTaint = args[0];
        if (sourceTaint == null) {
            throw new IllegalArgumentException(
                    "No source taint provided in environment variable args[0].");
        }
        try {
            value = Short.parseShort(sourceTaint);
        } catch (NumberFormatException e) {
            throw new IllegalArgumentException(
                    "Source taint is not a valid short.", e);
        }
        weakness(value, System.err);
    }
```

Approved for public release; distribution unlimited.

```
}
```

J - CWE-194A Source Code

```
package com.pontetec.weakness.numberhandling;

import java.io.PrintStream;
import com.pontetec.stonesoup.trace.Tracer;

public class CWE194A {

    public static void weakness(final short value, PrintStream output) {
        /*
         * CWE-194 Unexpected Sign Extension
         * This test takes in a value from a taint source,
         * which is then used to allocate an array. This value is
         * positive when stored in a byte, but if the value is between 128 and
         * 255, becomes negative if stored in a byte. When this negative value
         * is stored in a larger container (an int) it becomes a very large
         * number. When this number is used as part of a loop, it results in an
         * out of bounds exception.
         */
        // Good values: 1 - 127
        // Bad values: 128 - 255 */

        Tracer.tracepointWeaknessStart("CWE194", "A", "Unexpected Sign Extension");

        short stonesoup_array_size = value;

        Tracer.tracepointVariableShort("stonesoup_array_size", stonesoup_array_size);

        if (stonesoup_array_size < 0) {
            stonesoup_array_size = 0;
        } else if (stonesoup_array_size > 255) {
            stonesoup_array_size = 255;
        }

        Tracer.tracepointVariableShort("stonesoup_array_size", stonesoup_array_size);

        /* cast value will become negative for all input values greater than 127 */
        byte stonesoup_counter_max_signed = (byte)stonesoup_array_size;

        Tracer.tracepointVariableByte("stonesoup_counter_max_signed",
stonesoup_counter_max_signed);

        /* array to initialize */
        int[] stonesoup_array = new int[stonesoup_array_size];
        Tracer.tracepointBufferInfo("stonesoup_array", stonesoup_array.length, "Length
of stonesoup_array");

        Tracer.tracepointMessage("CROSSOVER-POINT: BEFORE");
        /* STONESOUP: crossover point (promotion to unsigned causes unexpected sign
extension) */
        char stonesoup_counter_max = (char) stonesoup_counter_max_signed;
        Tracer.tracepointVariableChar("stonesoup_counter_max", stonesoup_counter_max);
```

```
        Tracer.tracepointMessage("CROSSOVER-POINT: AFTER");

        try {
            /* initialize the array to all ones */
            Tracer.tracepointMessage("TRIGGER-POINT: BEFORE");
            for (char counter = 0; counter < stonesoup_counter_max; counter++) {
                /* STONESOUP: trigger point (sign extension causes out of bounds
exception) */
                stonesoup_array[counter] = 1;
            }
            Tracer.tracepointMessage("TRIGGER-POINT: AFTER");
        } catch (java.lang.RuntimeException e) {
            Tracer.tracepointError(e.getClass().getName() + ": " + e.getMessage());
            e.printStackTrace(output);
            throw e;
        }

        Tracer.tracepointWeaknessEnd();
    }

    /**
     * @param args
     */
    public static void main(String[] args) {
        short value = 0;
        String sourcevalue = args[0];
        if (sourcevalue == null) {
            throw new IllegalArgumentException(
                    "No source value provided in environment variable args[0].");
        }
        try {
            value = Short.parseShort(sourcevalue);
        } catch (NumberFormatException e) {
            throw new IllegalArgumentException(
                    "Source taint is not a valid short.", e);
        }
        weakness(value, System.err);
    }
```

```
}
```

J - CWE-195A Source Code

```
package com.pontetec.weakness.numberhandling;

import java.io.PrintStream;
import com.pontetec.stonesoup.trace.Tracer;

public class CWE195A {
    public static void weakness(final short value,
            PrintStream output){
        /*
         * CWE-195 Signed to Unsigned Conversion Error
         * This test takes in a value passed as an argument
         * This value is originally signed, but is converted
         * to unsigned when stored in a char.  This value is then
         * used to index an allocated array with a signed value.
         */
        // Good values:  0 .. 32767, -32768
        // Bad values:    -1 .. -32767 */

        Tracer.tracepointWeaknessStart("CWE195", "A", "Signed to Unsigned Conversion
Error");

        Tracer.tracepointVariableShort("value", value);

        Tracer.tracepointMessage("CROSSOVER-POINT: BEFORE");
        /* STONESOUP: crossover point (user controlled value, convert from signed to
unsigned) */
        int[] stonesoup_array = new int[Math.abs(value)];
        char stonesoup_max_char = (char)((short)value);
        Tracer.tracepointBufferInfo("stonesoup_array", stonesoup_array.length, "Length
of stonesoup_array");
        Tracer.tracepointVariableChar("stonesoup_max_char", stonesoup_max_char);
        Tracer.tracepointMessage("CROSSOVER-POINT: AFTER");

        try {
            Tracer.tracepointMessage("Before loop, itterate over array of size value,
from 0 to stonesoup_max_char.");

            Tracer.tracepointMessage("TRIGGER-POINT: BEFORE");
            /* STONESOUP: trigger point (user controlled value) */
            for(char stonesoup_counter = 0; stonesoup_counter < stonesoup_max_char;
stonesoup_counter++) {
                output.printf("Counter value: \"%c\"\n", stonesoup_counter);
                stonesoup_array[stonesoup_counter] = 0;
            }
            Tracer.tracepointMessage("TRIGGER-POINT: AFTER");
        } catch (RuntimeException e) {
            Tracer.tracepointError(e.getClass().getName() + ": " + e.getMessage());
            e.printStackTrace(output);
            throw e;
        }
```

```
        Tracer.tracepointWeaknessEnd();
    }
     /**
     * @param args
     */
    public static void main(String[] args){
        short value = 0;
        String sourcevalue = args[0];
        if (sourcevalue == null) {
            throw new IllegalArgumentException("No source value provided in
environment variable args[0].");
        }
        try {
            value = Short.parseShort(sourcevalue);
        } catch (NumberFormatException e) {
            throw new IllegalArgumentException("Source taint is not a valid short.",
e);
        }
        weakness(value, System.err);
    }
```

```
}
```

J - CWE-196A Source Code

```
package com.pontetec.weakness.numberhandling;

import java.io.PrintStream;
import com.pontetec.stonesoup.trace.Tracer;

public class CWE196A {

    public static int[] stonesoupInitializeCounts(byte size) {
        Tracer.tracepointLocation("STONESOUP_INJECTED_FILE",
"stonesoupInitializeCounts");
        Tracer.tracepointVariableByte("size", size);

        if (size == 0) {
            return null;
        }

        Tracer.tracepointMessage("TRIGGER-POINT: BEFORE");
        /* STONESOUP: trigger point (user controlled value, convert from signed to
unsigned) */
        int[] result = new int[size];
        Tracer.tracepointMessage("TRIGGER-POINT: AFTER");
        Tracer.tracepointBufferInfo("result", result.length, "Length of result");

        for (int ii = 0; ii < result.length; ii++) {
            result[ii] = 0;
        }

        return result;
    }

    public static void weakness(final char value,
            PrintStream output){
        /*
         * CWE-196 Unsigned to Signed Conversion Error
         * This test takes in a char value from a taint source.
         * The value is cast to a byte causing an unsigned to
         * signed conversion. This value is then used to
         * allocate an array. If the original char is >127
         * this will lead to a negative index for the array
         * causing a NegativeArraySizeException.
         */
        // Good values: any char with value    0 - 127
        // Bad    values: any char with value  128 - 255

        Tracer.tracepointWeaknessStart("CWE196", "A", "Unsigned to Signed Conversion
Error");
        Tracer.tracepointVariableChar("value", value);

        try {
            Tracer.tracepointMessage("CROSSOVER-POINT: BEFORE");
            /* STONESOUP: crossover point (user controlled value, convert from
```

Approved for public release; distribution unlimited.

```
unsigned to signed) */
            int[] stonesoup_char_counts =
stonesoupInitializeCounts((byte)((char)value));
            Tracer.tracepointMessage("CROSSOVER-POINT: AFTER");

            for (char counter = 0; counter < value; counter++) {
                stonesoup_char_counts[counter] += 1;
            }
            Tracer.tracepointBufferInfo("stonesoup_char_counts",
stonesoup_char_counts.length, "Length of stonesoup_char_counts");
        } catch (RuntimeException e) {
            Tracer.tracepointError(e.getClass().getName() + ": " + e.getMessage());
            e.printStackTrace(output);
            throw e;
        }

        Tracer.tracepointWeaknessEnd();
    }
    /**
     * @param args
     */
    public static void main(String[] args){
        char value = 0;
        String sourcevalue = args[0];
        if (sourcevalue == null) {
            throw new IllegalArgumentException("No source value provided in
environment variable args[0].");
        }
        else if (sourcevalue.length() == 0) {
            throw new IllegalArgumentException("No source value provided in
environment variable args[0].");
        }
        try {
            value = sourcevalue.charAt(0);
        } catch (NumberFormatException e) {
            throw new IllegalArgumentException("Source taint is not a valid short.",
e);
        }
        weakness(value, System.err);
    }
```

```
}
```

J - CWE-197A Source Code

```
package com.pontetec.weakness.numberhandling;

import java.io.PrintStream;
import java.security.NoSuchAlgorithmException;
import java.security.SecureRandom;
import com.pontetec.stonesoup.trace.Tracer;

public class CWE197A {
    public static void weakness(final long value, PrintStream output) {
        /*
         * CWE-197 Numeric Truncation Error
         * This test takes in a value from a taint source.
         * The value is truncated to an integer, and then used to generate
         * a random number within a range.    If the converted integer value
         * is negative, SecureRandom.nextInt(value) will throw a
         * RuntimeException.
         */
        // Good values: 0 - 2,147,483,647
        // Bad values: >= 2,147,483,648

        Tracer.tracepointWeaknessStart("CWE197", "A", "Numeric Trucation Error");

        Tracer.tracepointVariableLong("value", value);

        if (value > 0) {
            Tracer.tracepointMessage("CROSSOVER-POINT: BEFORE");
            /* STONESOUP: crossover point (user controlled value, convert from long to
int) */
            int stonesoup_max_value = (int)((long)value);
            Tracer.tracepointMessage("CROSSOVER-POINT: AFTER");

            Tracer.tracepointVariableInt("stonesoup_max_value", stonesoup_max_value);


            SecureRandom random = null;
            try {
                Tracer.tracepointMessage("Creating PRNG.");
                random = SecureRandom.getInstance("SHA1PRNG");
            } catch (NoSuchAlgorithmException e) {
                output.println("STONESOUP: Failed to create PRNG.");
                e.printStackTrace(output);
            }

            if (random != null) {
                Tracer.tracepointMessage("Generating random variable between 0 and
stonesoup_max_value");
                try {
                    output.printf("Generating random value between %d (inclusive) and
%d (exclusive).\n", 0, stonesoup_max_value);
                    Tracer.tracepointMessage("TRIGGER-POINT: BEFORE");
                    /* STONESOUP: trigger point (value is now negative, causing
```

```
exception) */
                    output.printf("Random choice: %d\n",
random.nextInt(stonesoup_max_value));
                    Tracer.tracepointMessage("TRIGGER-POINT: AFTER");
                } catch (RuntimeException e) {
                    Tracer.tracepointError(e.getClass().getName() + ": " +
e.getMessage());
                    e.printStackTrace(output);
                    throw e;
                }
                Tracer.tracepointMessage("After random value generation.");
            }
        }
        Tracer.tracepointWeaknessEnd();
    }

    /**
     * @param args
     */
    public static void main(String[] args) {
        long value = 0;
        String sourceTaint = args[0];
        if (sourceTaint == null) {
            throw new IllegalArgumentException(
                    "No source taint provided in environment variable args[0].");
        }
        try {
            value = Long.parseLong(sourceTaint);
        } catch (NumberFormatException e) {
            throw new IllegalArgumentException(
                    "Source taint is not a valid long.", e);
        }
        weakness(value, System.err);
    }
```

```
}
```

J - CWE-209A Source Code

```
package com.pontetec.weakness.errorhandling;

import java.io.PrintStream;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.sql.Types;
import com.pontetec.stonesoup.trace.Tracer;

public class CWE209A {
    public static void weakness(final String companyName, PrintStream output) {
        /* CWE-209 Information Exposure Through an Error Message
         *
         * This test attempts to insert a new record into a MySQL database.
         * In the event of a SQLException during execution of the insert
         * command, diagnostic information about the database connection
         * and SQL statement will be dumped to the output stream.
         *
         * Goods: Any string <= 40 characters
         * Bads:  Any string > 40 characters
         */

        Tracer.tracepointWeaknessStart("CWE209", "A", "Information Exposure Through an
Error Message");

        String stonesoup_mysql_host = System.getenv("DBMYSQLHOST");
        String stonesoup_mysql_port = System.getenv("DBMYSQLPORT");
        String stonesoup_mysql_user = System.getenv("DBMYSQLUSER");
        String stonesoup_mysql_pass = System.getenv("DBMYSQLPASSWORD");
        String stonesoup_mysql_dbname = System.getenv("SS_DBMYSQLDATABASE");

        Tracer.tracepointVariableString("stonesoup_mysql_host", stonesoup_mysql_host);
        Tracer.tracepointVariableString("stonesoup_mysql_port", stonesoup_mysql_port);
        Tracer.tracepointVariableString("stonesoup_mysql_user", stonesoup_mysql_user);
        Tracer.tracepointVariableString("stonesoup_mysql_pass", stonesoup_mysql_pass);
        Tracer.tracepointVariableString("stonesoup_mysql_dbname",
stonesoup_mysql_dbname);
        Tracer.tracepointVariableString("companyName", companyName);

        if (stonesoup_mysql_host == null || stonesoup_mysql_port == null ||
                stonesoup_mysql_user == null || stonesoup_mysql_pass == null ||
                stonesoup_mysql_dbname == null) {
            Tracer.tracepointError("Missing required database connection
parameter(s).");
            output.println("STONESOUP: Missing required DB connection parameters.");
        } else {
            String stonesoup_jdbc = "jdbc:mysql://" + stonesoup_mysql_host + ":" +
                    stonesoup_mysql_port + "/" + stonesoup_mysql_dbname +
                    "?dumpQueriesOnException=true";
```

```
            Tracer.tracepointVariableString("stonesoup_jdbc", stonesoup_jdbc);

            if (companyName == null) {
                output.println("No company name provided.");
            } else {
                Connection con = null;
                try {
                    /* load the mysql driver */
                    Class.forName("com.mysql.jdbc.Driver");

                    con = DriverManager.getConnection(
                            stonesoup_jdbc, stonesoup_mysql_user,
stonesoup_mysql_pass);

                    try {
                        /* rather than building a string, use a prepared statement
with params */
                        PreparedStatement stmt = con.prepareStatement("INSERT INTO
Shippers (CompanyName, Phone) VALUES (?, ?)");

                        Tracer.tracepointMessage("CROSSOVER-POINT: BEFORE");
                        /* STONESOUP: crossover point (user controlled value in query)
*/
                        stmt.setString(1, companyName);
                        stmt.setNull(2, Types.NULL);
                        Tracer.tracepointMessage("CROSSOVER-POINT: AFTER");

                        Tracer.tracepointMessage("TRIGGER-POINT: BEFORE");
                        /* STONESOUP: trigger point */
                        if (stmt.executeUpdate() > 0) {
                            output.println("Shipper added successfully.");
                        } else {
                            output.println("No rows added.");
                        }
                        Tracer.tracepointMessage("TRIGGER-POINT: AFTER");
                    } catch (SQLException se) {
                        Tracer.tracepointError("SQLException: Printing connection
details");
                        output.println("Database Error!");
                        output.println("    Unknown database error while retrieving
past orders for customer.");
                        output.println("");
                        output.println("Connection Details");
                        output.printf("    Host: %s\n", stonesoup_mysql_host);
                        output.printf("    Port: %s\n", stonesoup_mysql_port);
                        output.printf("    User: %s\n", stonesoup_mysql_user);
                        output.printf("    Pass: %s\n", stonesoup_mysql_pass);
                        output.printf("    JDBC: %s\n", stonesoup_jdbc);
                        output.println("");
                        output.println("Error Message");
                        output.println(se.getMessage());
                        output.println("");
                        output.println("Stacktrace");
                        se.printStackTrace(output);
                    }
                } catch (SQLException se) {
                    Tracer.tracepointError(se.getClass().getName() + ": " +
se.getMessage());
                    output.println("STONESOUP: Failed to connect to DB.");
```

```
                    se.printStackTrace(output);
                } catch (ClassNotFoundException cnfe) {
                    Tracer.tracepointError(cnfe.getClass().getName() + ": " +
cnfe.getMessage());
                    output.println("STONESOUP: Failed to load DB driver.");
                    cnfe.printStackTrace(output);
                } finally {
                    /* attempt to close the connection cleanly */
                    try {
                        if (con != null && !con.isClosed()) {
                            con.close();
                        }
                    } catch (SQLException e) {
                        Tracer.tracepointError(e.getClass().getName() + ": " +
e.getMessage());
                        output.println("STONESOUP: Failed to close DB connection.");
                        e.printStackTrace(output);
                    }
                }
            }
        }
        Tracer.tracepointWeaknessEnd();
    }

    /**
     * @param args
     */
    public static void main(String[] args) {
        String sourceTaint = args[0];
        if (sourceTaint == null) {
                throw new IllegalArgumentException(
                                "No source taint provided in environment variable
args[0].");
        }
       weakness(sourceTaint, System.err);
    }
```

Approved for public release; distribution unlimited.

```
}
```

J - CWE-248A Source Code

```
package com.pontetec.weakness.errorhandling;

import java.io.PrintStream;
import java.lang.Character;
import com.pontetec.stonesoup.trace.Tracer;

public class CWE248A {
    public static void weakness(final String value,
            PrintStream output) {
        /*
         * CWE-248 Uncaught Exception
         *
         * This test case takes a value and attempts to convert it to
         * upper case.    If the value contains anything other than a-zA-Z,
         * then an exception is thrown, but not caught.
         *
         * GOOD Values - a-zA-Z
         * BAD Values - 0-9.?!@#$%^
         */
        Tracer.tracepointWeaknessStart("CWE248", "A", "Uncaught Exception");

        final String stonesoup_input_for_print = value;

        Tracer.tracepointVariableString("value", value);

        Thread.UncaughtExceptionHandler stonesoup_ueh = Thread
                .getDefaultUncaughtExceptionHandler();
        Thread.setDefaultUncaughtExceptionHandler(new
Thread.UncaughtExceptionHandler() {
            // @Override - removed due to Java 1.4
            public void uncaughtException(Thread t, Throwable e) {
                Tracer.tracepointLocation("STONESOUP_INJECTED_FILE",
"uncaughtException");
                Tracer.tracepointMessage("CROSSOVER-POINT: AFTER");
                Tracer.tracepointMessage("TRIGGER-POINT: BEFORE");
                /* STONESOUP: TRIGGER POINT */
                System.err.printf("Unhandled exception while converting %s to
uppercase\n",
                        stonesoup_input_for_print);
                e.printStackTrace(System.err);
                Tracer.tracepointMessage("TRIGGER-POINT: AFTER");
            }
        });

        char[] stonesoup_characters = value.toCharArray();
        for(int i=0; i < stonesoup_characters.length; i++){
            if(Character.isLowerCase(stonesoup_characters[i])){

stonesoup_characters[i]=Character.toUpperCase(stonesoup_characters[i]);
            }
            else if(!Character.isUpperCase(stonesoup_characters[i])){
```

```
                Tracer.tracepointMessage("CROSSOVER-POINT: BEFORE");
                /* STONESOUP: CROSSOVER POINT */
                throw new java.lang.IllegalArgumentException("Not a valid character");
            }
        }
        Tracer.tracepointVariableString("stonesoup_characters",
String.valueOf(stonesoup_characters));
        output.println(String.valueOf(stonesoup_characters));
        Thread.setDefaultUncaughtExceptionHandler(stonesoup_ueh);
        Tracer.tracepointWeaknessEnd();
    }

    /**
     * @param args
     */
    public static void main(String[] args) {
        String sourceTaint = args[0];
        if (sourceTaint == null) {
                throw new IllegalArgumentException(
                                "No source taint provided in environment variable
args[0].");
        }
       weakness(sourceTaint, System.err);
    }
}
```

J - CWE-252A Source Code

```
package com.pontetec.weakness.errorhandling;

import java.io.PrintStream;
import com.pontetec.stonesoup.trace.Tracer;

public class CWE252A {
    public static String stonesoup_to_upper(final String input) {
        Tracer.tracepointLocation("STONESOUP_INJECTED_FILE", "stonesoup_to_upper");
        char stonesoup_char = 0;
        String retval = input;
        for (int i = 0; i < retval.length(); i++) {
            stonesoup_char = retval.charAt(i);
            if (Character.isLowerCase(stonesoup_char)) {
                retval = retval.replace(stonesoup_char,
                        Character.toUpperCase(stonesoup_char));
            } else if (!Character.isUpperCase(stonesoup_char)) {
                return null;
            }
        }
        return retval;
    }

    public static void weakness(final String value, PrintStream output) {
        /*
         * CWE-252 Unchecked Return Value
         *
         * This weakness takes a value and converts it to upper case.
         * If it contains characters not a-zA-Z, then the function
         * used to covert to upper case will return null.  After
         * converting the value to upper case it is compared to a static
         * 'password'.    This comparison will cause an error if the
         * value contained a number, punctuation, space, etc.
         *
         * GOOD Values - a-zA-Z
         * BAD Values - 0-9.?!@#$%^
         */

        Tracer.tracepointWeaknessStart("CWE252", "A", "Unchecked Return Value");

        Tracer.tracepointMessage("CROSSOVER-POINT: BEFORE");
        /* STONESOUP: CROSSOVER POINT */
        String capitalized_value = stonesoup_to_upper(value);
        Tracer.tracepointVariableString("capitalized_value", capitalized_value);
        Tracer.tracepointMessage("CROSSOVER-POINT: AFTER");

        String password = "STONESOUP";
        try {
            Tracer.tracepointMessage("TRIGGER-POINT: BEFORE");
            /* STONESOUP: TRIGGER POINT */
            if(    password.compareTo(capitalized_value)==0){
                output.println("passwords match");
```

```
            }
            else {
                output.println("passwords don't match");
            }
            Tracer.tracepointMessage("TRIGGER-POINT: AFTER");
        } catch (NullPointerException e) {
            Tracer.tracepointError(e.getClass().getName() + ": " + e.getMessage());
            e.printStackTrace(output);
            throw e;
        }
        Tracer.tracepointWeaknessEnd();
    }

    /**
     * @param args
     */
    public static void main(String[] args) {
        String sourceTaint = args[0];
        if (sourceTaint == null) {
                throw new IllegalArgumentException(
                                "No source taint provided in environment variable
args[0].");
        }
       weakness(sourceTaint, System.err);
    }
}
```

J - CWE-252B Source Code

```
package com.pontetec.weakness.errorhandling;

import java.io.PrintStream;
import com.pontetec.stonesoup.trace.Tracer;

public class CWE252B {

    public static void weakness(final String stonesoup_ss_sensitive_file,
            PrintStream output) {
        /*
         * CWE-252 Unchecked Return
         *
         * This test reads data from a "sensitive" file without checking the
         * return value, and then repeats the operation on a non-sensitive file,
         * then printing the second file. Technical Impact is information
         * leakage.
         *
         * Inputs: ss_sensitive_file - filename of a file to read.
         * SS_OTHER_FILE - filename of a second file to read.
         * Good Inputs: any combination where the length of SS_OTHER_FILE
         *                 is greater than the length of the SS_SENSITIVE_FILE.
         * Bad Inputs: any combination where the length of SS_OTHER_FILE
         * is less than the length of the SS_SENSITIVE_FILE.
         *
         */
        Tracer.tracepointWeaknessStart("CWE252", "B", "Unchecked Return");

        try {
            final int STONESOUP_BUFFER_SIZE = 2048;

            String stonesoup_sensitiveFName = stonesoup_ss_sensitive_file;
            String stonesoup_otherFName = System.getenv("SS_OTHER_FILE");
            byte[] stonesoup_buff = new byte[STONESOUP_BUFFER_SIZE];

            Tracer.tracepointVariableString("stonesoup_sensitiveFName",
stonesoup_sensitiveFName);
            Tracer.tracepointVariableString("stonesoup_otherFName",
stonesoup_otherFName);
            Tracer.tracepointBufferInfo("stonesoup_buff", stonesoup_buff.length,
"Length of stonesoup_buff");

            java.io.InputStream stonesoup_sensitiveFile = new java.io.FileInputStream(
                    stonesoup_sensitiveFName);
            java.io.InputStream stonesoup_otherFile = new java.io.FileInputStream(
                    stonesoup_otherFName);

            Tracer.tracepointMessage("CROSSOVER-POINT: BEFORE");
            /* read ignoring return value */
            /* STONESOUP: CROSSOVER POINT */
            stonesoup_sensitiveFile.read(stonesoup_buff);
            stonesoup_sensitiveFile.close();
```

```
            Tracer.tracepointMessage("CROSSOVER-POINT: AFTER");

            Tracer.tracepointMessage("TRIGGER-POINT: BEFORE");
            /* now read other file, which is 1 character */
            /* STONESOUP: TRIGGER POINT */
            stonesoup_otherFile.read(stonesoup_buff);
            stonesoup_otherFile.close();
            Tracer.tracepointMessage("TRIGGER-POINT: AFTER");

            /* print buffer */
            String output_data = new String(stonesoup_buff);
            Tracer.tracepointVariableString("output_data", output_data);
            output.println("Output is:\n" + output_data);

        } catch (java.io.IOException ioe) {
            Tracer.tracepointError(ioe.getClass().getName() + ": " +
ioe.getMessage());
            output.println("STONESOUP: Error accessing files");
            ioe.printStackTrace(output);
        }
        Tracer.tracepointWeaknessEnd();
    }

    /**
     * @param args
     */
    public static void main(String[] args) {
        String sourceTaint = args[0];
        if (sourceTaint == null) {
                throw new IllegalArgumentException(
                                "No source taint provided in environment variable
args[0].");
        }
       weakness(sourceTaint, System.err);
    }
}
```

J - CWE-253A Source Code

```
package com.pontetec.weakness.errorhandling;

import java.io.PrintStream;
import com.pontetec.stonesoup.trace.Tracer;

public class CWE253A {

    public static void weakness(final String stonesoup_tainted_buff, PrintStream
output) {
        /*
         * CWE-253 Incorrect Check of Function Return Value
         * This test gets the index of the character '.' in the input string.
         * It then uses that index to get a substring beginning with '.'
         * If '.' does not occur in the string, the index will be -1, and then
         * the substring operation will fail with an IndexOutOfBounds exception.
         */
        Tracer.tracepointWeaknessStart("CWE253", "A", "Incorrect Check of Function
Return Value");

        int location = stonesoup_tainted_buff.indexOf('.');

        Tracer.tracepointVariableString("stonesoup_tainted_buff",
stonesoup_tainted_buff);
        Tracer.tracepointVariableInt("location", location);

        Tracer.tracepointMessage("CROSSOVER-POINT: BEFORE");
        /* STONESOUP: CROSSOVER POINT */
        if (location != 0) {
            Tracer.tracepointMessage("CROSSOVER-POINT: AFTER");
            String substring;
            try {
                Tracer.tracepointMessage("TRIGGER-POINT: BEFORE");
                /* STONESOUP: TRIGGER POINT */
                substring = stonesoup_tainted_buff.substring(location);
                Tracer.tracepointVariableString("substring", substring);
                Tracer.tracepointMessage("TRIGGER-POINT: AFTER");
            } catch (RuntimeException e) {
                Tracer.tracepointError(e.getClass().getName() + ": " +
e.getMessage());
                e.printStackTrace(output);
                throw e;
            }
            output.println("Substring beginning with '.' is \"" + substring + "\"\n");
        } else {
            Tracer.tracepointMessage("CROSSOVER-POINT: AFTER");
            output.println("'.' appears at start of line\n");
        }
        Tracer.tracepointWeaknessEnd();
    }

    /**
```

```
     * @param args
     */
    public static void main(String[] args){
        String sourceTaint = args[0];
        if (sourceTaint == null) {
                throw new IllegalArgumentException(
                                "No source taint provided in environment variable
args[0].");
        }
        weakness(sourceTaint, System.err);
    }
}
```

J - CWE-363A Source Code

```
package com.pontetec.weakness.concurrency;

import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.io.File;
import java.io.PrintStream;
import java.io.PrintWriter;
import com.pontetec.stonesoup.trace.Tracer;

public class CWE363A {
    /*
     * "This test takes an integer followed by a filename as input
     * (int filename), and checks that the file exists, is in the current directory,
and is
     * not a symbolic link. However the weakness takes some time before opening the
file
     * allowing a time of check, time of use vulnerability. This vulnerability allows
an
     * attacker to replace the file with a symbolic link to a file outside of the
current
     * directory during the time between checking the file's validity and opening it.
     */

    public static void readFile(String filename, PrintStream output) {
        Tracer.tracepointLocation("STONESOUP_INJECTED_FILE", "readFile");
        String str;
        try {
            BufferedReader reader = new BufferedReader(new FileReader(filename));
            while ((str = reader.readLine()) != null) {
                output.println(str);
            }
            reader.close();
        } catch (FileNotFoundException e) {
            Tracer.tracepointError("Error reading syncFile.  " +
e.getClass().getName() + ": " + e.getMessage());
            output.println("Error reading sync file: " + e);
        } catch (IOException e) {
            Tracer.tracepointError("Error reading syncFile.  " +
e.getClass().getName() + ": " + e.getMessage());
            output.println("Error reading sync file: " + e);
        }
    }

    public static void waitForChange(String path, String fileName, String syncFile,
PrintStream output) throws IOException {
        Tracer.tracepointLocation("STONESOUP_INJECTED_FILE", "waitForChange");

        PrintWriter writer = new PrintWriter(path + fileName + ".pid");
        writer.close();
```

```
        Tracer.tracepointVariableString(".pid file", path + fileName + ".pid");

        Tracer.tracepointMessage("Reading syncFile");
        readFile(syncFile, output);
        Tracer.tracepointMessage("Finished reading syncFile");
    }

    public static boolean isValidPath(String file) {
        Tracer.tracepointLocation("STONESOUP_INJECTED_FILE", "isValidPath");
        return !file.contains("/");
    }

    public static boolean isSymlink(File file) throws IOException {
        Tracer.tracepointLocation("STONESOUP_INJECTED_FILE", "isSymlink");
        return !file.getCanonicalFile().equals(file.getAbsoluteFile());
    }

    public static void weakness(final String stonesoup_value, PrintStream
stonesoup_output) {
        Tracer.tracepointWeaknessStart("CWE363", "A", "Race Condition Enabling Link
Following");

        String stonesoup_syncfile = null;
        String stonesoup_fileName = null;

        String stonesoup_substrings[] = stonesoup_value.split("\\s+", 2);

        if(stonesoup_substrings.length == 2){

            try{
                stonesoup_syncfile = stonesoup_substrings[0];
                stonesoup_fileName = stonesoup_substrings[1];

                Tracer.tracepointVariableString("stonesoup_value", stonesoup_value);
                Tracer.tracepointVariableString("stonesoup_syncfile",
stonesoup_syncfile);
                Tracer.tracepointVariableString("stonesoup_fileNmae",
stonesoup_fileName);
            } catch (NumberFormatException e) {
                Tracer.tracepointError(e.getClass().getName() + ": " +
e.getMessage());
                stonesoup_output.println("NumberFormatException");
            }

            if (isValidPath(stonesoup_fileName)) {
                String stonesoup_path="/opt/stonesoup/workspace/testData/";
                File stonesoup_file = new File(stonesoup_path, stonesoup_fileName);
                BufferedReader stonesoup_reader = null;
                String stonesoup_line = "";

                Tracer.tracepointVariableString("stonesoup_file",
stonesoup_file.getAbsolutePath());

                if (stonesoup_file.exists()) {
                    try {
                        if (!isSymlink(stonesoup_file)) {
                            Tracer.tracepointMessage("CROSSOVER-POINT: BEFORE");
                            /* STONESOUP: CROSSOVER POINT (Race Condition Enabling
```

```
Link Following) */
                            waitForChange(stonesoup_path, stonesoup_fileName,
stonesoup_syncfile, stonesoup_output);
                            Tracer.tracepointMessage("CROSSOVER-POINT: AFTER");
                            Tracer.tracepointMessage("TRIGGER-POINT: BEFORE");
                            /* STONESOUP: TRIGGER POINT (Race Condition Enabling Link
Following) */
                            stonesoup_reader = new BufferedReader(new
FileReader(stonesoup_file.getAbsolutePath()));
                            while ((stonesoup_line = stonesoup_reader.readLine()) !=
null) {
                                stonesoup_output.println(stonesoup_line);
                            }
                            stonesoup_reader.close();
                            Tracer.tracepointMessage("TRIGGER-POINT: AFTER");
                        }
                        else {
                            Tracer.tracepointError("ERROR: File is a symlink!");
                            stonesoup_output.println("ERROR: File is a symlink!");
                        }
                    }
                    catch(IOException e) {
                        Tracer.tracepointError("ERROR: File got deleted.");
                        stonesoup_output.println("ERROR: File got deleted.");
                    }
                }
                else {
                    Tracer.tracepointError("ERROR: File doesn't exist!");
                    stonesoup_output.println("ERROR: File doesn't exist!");
                }
            }
            else {
                Tracer.tracepointError("ERROR: Filename isn't valid!");
                stonesoup_output.println("ERROR: Filename isn't valid!");
            }
        } else {
            Tracer.tracepointError("ERROR: Input isn't valid!");
            stonesoup_output.println("ERROR: Input isn't valid!");
        }
        Tracer.tracepointWeaknessEnd();
    }

    public static void main(String[] args) {
        if (args.length == 0) {
            throw new IllegalArgumentException(
                    "No arguments provided on command-line.");
        }
        String sourceTaint = args[0];
        if (sourceTaint == null) {
            throw new IllegalArgumentException(
                    "No source taint provided in environment variable args[0].");
        }
        weakness(sourceTaint, System.err);
    }
```

```
}
```

J - CWE-367A Source Code

```
package com.pontetec.weakness.concurrency;

import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.io.File;
import java.io.PrintStream;
import java.io.PrintWriter;
import com.pontetec.stonesoup.trace.Tracer;

public class CWE367A {
    /*
     * "This test takes an integer and filename as input
     * (int filename), and checks that the file exists, and is in the current
directory.
     * However there is a  time of check,time of use vulnerability after the file is
checked
     * but before it is opened allowing the file to be deleted before opening causing
a null
     * pointer dereference.\n";
     */

    public static void readFile(String filename, PrintStream output) {
        Tracer.tracepointLocation("STONESOUP_INJECTED_FILE", "readFile");
        String str;
        try {
            BufferedReader reader = new BufferedReader(new FileReader(filename));
            while ((str = reader.readLine()) != null) {
                output.println(str);
            }
            reader.close();
        } catch (FileNotFoundException e) {
            Tracer.tracepointError("Error reading syncFile.  " +
e.getClass().getName() + ": " + e.getMessage());
            output.println("Error reading sync file: " + e);
        } catch (IOException e) {
            Tracer.tracepointError("Error reading syncFile.  " +
e.getClass().getName() + ": " + e.getMessage());
            output.println("Error reading sync file: " + e);
        }
    }

    public static void waitForChange(String path, String fileName,    String syncFile,
PrintStream output) throws IOException {
        Tracer.tracepointLocation("STONESOUP_INJECTED_FILE", "waitForChange");

        PrintWriter writer = new PrintWriter(path + fileName + ".pid");
        writer.close();

        Tracer.tracepointMessage("Reading syncFile");
```

```
        readFile(syncFile, output);
        Tracer.tracepointMessage("Finished reading syncFile");
    }

    public static boolean isValidPath(String file) {
        return !file.contains("/");
    }

    public static void weakness(final String stonesoup_value, PrintStream
stonesoup_output) {
        Tracer.tracepointWeaknessStart("CWE367", "A", "Time-of-check Time-of-use
(TOCTOU) Race Condition");

        String stonesoup_syncfile = null;
        String stonesoup_fileName = null;

        String stonesoup_substrings[] = stonesoup_value.split("\\s+", 2);

        if(stonesoup_substrings.length == 2){

            try{
                stonesoup_syncfile = stonesoup_substrings[0];
                stonesoup_fileName = stonesoup_substrings[1];

                Tracer.tracepointVariableString("stonesoup_value", stonesoup_value);
                Tracer.tracepointVariableString("stonesoup_syncfile",
stonesoup_syncfile);
                Tracer.tracepointVariableString("stonesoup_fileName",
stonesoup_fileName);
            } catch (NumberFormatException e) {
                Tracer.tracepointError(e.getClass().getName() + ": " +
e.getMessage());
                stonesoup_output.println("NumberFormatException");
            }

            String stonesoup_line = "";
            File stonesoup_file = null;
            BufferedReader stonesoup_reader = null;

            String stonesoup_path="/opt/stonesoup/workspace/testData/";

            if (isValidPath(stonesoup_fileName)) {
                stonesoup_file = new File(stonesoup_path, stonesoup_fileName);
                if (stonesoup_file.exists()) {
                    try {
                        Tracer.tracepointMessage("CROSSOVER-POINT: BEFORE");
                        /* STONESOUP: CROSS-OVER POINT (time of check, time of use) */
                        waitForChange(stonesoup_path, stonesoup_fileName,
stonesoup_syncfile, stonesoup_output);
                        Tracer.tracepointMessage("CROSSOVER-POINT: AFTER");

                        Tracer.tracepointMessage("TRIGGER-POINT: BEFORE");
                        /* STONESOUP: TRIGGER POINT (time of check, time of use) */
                        stonesoup_reader = new BufferedReader(new
FileReader(stonesoup_file.getAbsolutePath()));
                        while ((stonesoup_line = stonesoup_reader.readLine()) != null)
{
                            stonesoup_output.println(stonesoup_line);
                        }
```

```
                        stonesoup_reader.close();
                        Tracer.tracepointMessage("TRIGGER-POINT: AFTER");
                    }
                    catch(IOException e) {
                        Tracer.tracepointError(e.getClass().getName() + ": " +
e.getMessage());
                        e.printStackTrace(stonesoup_output);
                    }
                }
                else {
                    Tracer.tracepointError("File doesn't exist!");
                    stonesoup_output.println("ERROR: File doesn't exist!");
                }
            }
            else {
                Tracer.tracepointError("Filename isn't valid!");
                stonesoup_output.println("ERROR: Filename isn't valid!");
            }
        }
        else {
            Tracer.tracepointError("Input isn't valid!");
            stonesoup_output.println("ERROR: Input isn't valid!");
        }
        Tracer.tracepointWeaknessEnd();
    }

    public static void main(String[] args) {
        if (args.length == 0) {
            throw new IllegalArgumentException(
                    "No arguments provided on command-line.");
        }
        String sourceTaint = args[0];
        if (sourceTaint == null) {
            throw new IllegalArgumentException(
                    "No source taint provided in environment variable args[0].");
        }
        weakness(sourceTaint, System.err);
    }
```

```
}
```

J - CWE-369A Source Code

```
package com.pontetec.weakness.numberhandling;

import java.io.PrintStream;
import com.pontetec.stonesoup.trace.Tracer;

public class CWE369A {
    public static void weakness(final int value, PrintStream output) {
        /*
         * CWE-369 Divide By Zero
         * This test takes in a value passed in as an
         * argument and uses it to compute a divisor which is
         * later used in a modulus operation.
         */
        // Good values: 1 - 32767, 32769 - 65535, etc.
        // Bad values: any multiple of 32768

        Tracer.tracepointWeaknessStart("CWE369", "A", "Divide By Zero");

        Tracer.tracepointVariableInt("value", value);

        if (value != 0) {
            try {
                Tracer.tracepointMessage("CROSSOVER-POINT: BEFORE");
                /* STONESOUP: crossover point (Multiple of 32768 results in zero
divisor) */
                int random = (8191 * value) % (1 << 15);
                Tracer.tracepointVariableInt("random", random);
                Tracer.tracepointMessage("CROSSOVER-POINT: AFTER");

                Tracer.tracepointMessage("TRIGGER-POINT: BEFORE");
                /* STONESOUP: trigger point (Multiple of 32768 generates divide by
zero) */
                int factor = (1 << 31) % random;
                Tracer.tracepointVariableInt("factor", factor);
                Tracer.tracepointMessage("TRIGGER-POINT: AFTER");

                output.printf("Random Factor: %d\n", factor);
            } catch (java.lang.RuntimeException e) {
                Tracer.tracepointError(e.getClass().getName() + ": " +
e.getMessage());
                e.printStackTrace(output);
                throw e;
            }
        }
        Tracer.tracepointWeaknessEnd();
    }

    /**
     * @param args
     */
    public static void main(String[] args) {
```

```
        int value = 0;
        String sourceTaint = args[0];
        if (sourceTaint == null) {
            throw new IllegalArgumentException(
                    "No source taint provided in environment variable args[0].");
        }
        try {
            value = Integer.parseInt(sourceTaint);
        } catch (NumberFormatException e) {
            throw new IllegalArgumentException(
                    "Source taint is not a valid integer.", e);
        }
        weakness(value, System.err);
    }
```

```
}
```

J - CWE-390A Source Code

```
package com.pontetec.weakness.errorhandling;

import java.io.IOException;
import java.io.PrintStream;
import com.pontetec.stonesoup.trace.Tracer;

public class CWE390A {
    public static void weakness(final String filename, PrintStream output) {
        /*
         * CWE-390 Detection of Error Condition Without Action
         * This test takes a filename and attemps to read and output
         * to the screen.  If the file does not exist, it swallows the
         * FileNotFoundException, and subsequently dies on a NullPointerException.
         */
        Tracer.tracepointWeaknessStart("CWE390", "A", "Detection of Error Condition
Without Action");

        Tracer.tracepointVariableString("filename", filename);

        java.io.FileInputStream fis = null;
        java.io.File readPath = new java.io.File(filename);

        Tracer.tracepointMessage("CROSSOVER-POINT: BEFORE");
        /* STONESOUP: CROSSOVER POINT (Detection of Error Condition Without Action) */
        try {
            fis = new java.io.FileInputStream(
                        readPath);
        } catch (java.io.FileNotFoundException e) {
        }
        Tracer.tracepointMessage("CROSSOVER-POINT: AFTER");

        try {
            Tracer.tracepointMessage("TRIGGER-POINT: BEFORE");
            /* STONESOUP: trigger point (read path can be any file) */
            java.io.BufferedReader reader = new java.io.BufferedReader(
                new java.io.InputStreamReader(fis));
            Tracer.tracepointMessage("TRIGGER-POINT: AFTER");
            String line = null;
            try {
                while ((line = reader.readLine()) != null) {
                    output.println(line);
                }
            } catch (IOException e) {
                Tracer.tracepointError(e.getClass().getName() + ": " +
e.getMessage());
                output.printf("Failed to read file.\n");
            }
        } catch (RuntimeException e) {
            Tracer.tracepointError(e.getClass().getName() + ": " + e.getMessage());
            e.printStackTrace(output);
            throw e;
```

```
        }
        Tracer.tracepointWeaknessEnd();
    }

    /**
     * @param args
     */
    public static void main(String[] args){
        String sourceTaint = args[0];
        if (sourceTaint == null) {
                throw new IllegalArgumentException(
                                "No source taint provided in environment variable
args[0].");
        }
        weakness(sourceTaint, System.err);
    }
}
```

J - CWE-391A Source Code

```
package com.pontetec.weakness.errorhandling;

import java.io.PrintStream;
import com.pontetec.stonesoup.trace.Tracer;

public class CWE391A {
     public static void weakness(final int size, PrintStream output) {
        /*
         * CWE-391 Unchecked Error Condition
         * This test takes user input for the size of an array to be
         * allocated. If the array fails to allocate, an exception is
         * caught, but no action is taken resulting in a subsequent exception.
         */
        Tracer.tracepointWeaknessStart("CWE391", "A", "Unchecked Error Condition");

        int [] stonesoup_arr = null;

        Tracer.tracepointVariableInt("size", size);
        Tracer.tracepointMessage("CROSSOVER-POINT: BEFORE");
        try {
            output.printf("Allocating array of size %d\n", size);
            stonesoup_arr = new int[size];        // attempt to allocate array of size
'size'
        } catch (OutOfMemoryError e) {                      // catch an exception but do
nothing
            Tracer.tracepointError(e.getClass().getName() + ": " + e.getMessage());
            /* STONESOUP: CROSSOVER POINT (unchecked error condition) */
        }
        Tracer.tracepointBufferInfo("stonesoup_arr", (stonesoup_arr == null) ? 0 :
stonesoup_arr.length, "Length of stonesoup_arr");
        Tracer.tracepointMessage("CROSSOVER-POINT: AFTER");

        try {
            Tracer.tracepointMessage("TRIGGER-PONIT: BEFORE");
            /* STONESOUP: TRIGGER POINT (unchecked error condition) */
            for (int i = 0; i < stonesoup_arr.length; i++) {
                stonesoup_arr[i] = size - i;    // doesn't have space to allocate on
heap, errors out
            }
            Tracer.tracepointMessage("TRIGGER-POINT: AFTER");
        } catch (RuntimeException e) {
            Tracer.tracepointError(e.getClass().getName() + ": " + e.getMessage());
            e.printStackTrace(output);
            throw e;
        }
        Tracer.tracepointWeaknessEnd();
    }

     /**
     * @param args
     */
```

```
    public static void main(String[] args){
        int value = 0;
        String sourceTaint = args[0];
        if (sourceTaint == null) {
                throw new IllegalArgumentException(
                                "No source taint provided in environment variable
args[0].");
        }
        try {
                value = Integer.parseInt(sourceTaint);
        } catch (NumberFormatException e) {
                throw new IllegalArgumentException(
                                "Source taint is not a valid short.", e);
        }
        weakness(value, System.err);
    }
}
```

J - CWE-412A Source Code

```
package com.pontetec.weakness.concurrency;

import java.io.IOException;
import java.io.File;
import java.io.PrintStream;
import com.pontetec.stonesoup.trace.Tracer;

public class CWE412A {
    /*
     * This test takes a filename that is used as a representation of an externally
     * accessible and unrestricted mutex lock. The weakness will then attempt to
     * grab this lock by checking for the files existence, and creating the file
     * if it doesn't exist. If the file does exist, the weakness will hang until
     * it is deleted, indefinitely.
     */

    public static void weakness(final String stonesoup_value, PrintStream
stonesoup_output) {
        Tracer.tracepointWeaknessStart("CWE412", "A", "Unrestricted Externally
Accessible Lock");

        File stonesoup_file = null;
        String stonesoup_path="/opt/stonesoup/workspace/testData/";
        Tracer.tracepointVariableString("stonesoup_value", stonesoup_value);

        try {
            Tracer.tracepointMessage("CROSSOVER-POINT: BEFORE");
            /* STONESOUP: CROSS-OVER POINT (Unrestricted Externally Accessible Lock)
*/
            stonesoup_file = new File(stonesoup_path, stonesoup_value);
            Tracer.tracepointVariableString("stonesoup_path", stonesoup_path);
            Tracer.tracepointMessage("CROSSOVER-POINT: AFTER");

            Tracer.tracepointMessage("TRIGGER-POINT: BEFORE");
            /* STONESOUP: TRIGGER POINT (Unrestricted Externally Accessible Lock) */
            stonesoup_output.println(stonesoup_path);
            Tracer.tracepointMessage("Attempting to grab file lock.");
            while (!stonesoup_file.createNewFile()) {
                Thread.sleep(1);
            }
            Tracer.tracepointMessage("TRIGGER-POINT: AFTER");
            Tracer.tracepointMessage("Grabbed file lock.");
            stonesoup_output.println("File Created");
            stonesoup_file.delete();
        } catch (IOException e) {
            Tracer.tracepointError(e.getClass().getName() + ": " + e.getMessage());
            stonesoup_output.println("IOException");
        } catch (NullPointerException e) {
            Tracer.tracepointError(e.getClass().getName() + ": " + e.getMessage());
            stonesoup_output.println("NullPointerException");
        } catch (InterruptedException e) {
```

Approved for public release; distribution unlimited.

```
            Tracer.tracepointError(e.getClass().getName() + ": " + e.getMessage());
            stonesoup_output.println("InterruptedException");
        }
        Tracer.tracepointWeaknessEnd();
    }

    /**
     * @param args
     */
    public static void main(String[] args) {
        if (args.length == 0) {
            throw new IllegalArgumentException(
                    "No arguments provided on command-line.");
        }
        String sourceTaint = args[0];
        if (sourceTaint == null) {
            throw new IllegalArgumentException(
                    "No source taint provided in environment variable args[0].");
        }
        weakness(sourceTaint, System.err);
    }
```

```
}
```

J - CWE-414A Source Code

```
package com.pontetec.weakness.concurrency;

import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.io.PrintStream;
import java.util.Arrays;
import java.util.concurrent.locks.ReentrantLock;
import com.pontetec.stonesoup.trace.Tracer;

public class CWE414A {
    /*
     * This weakness takes a string in the form: '<qsize> <data>' where qsize
     * is the size of the array to sort (used to delay execution) and data is
     * a string that is used for processing. The contents of this string are
     * unimportant. Two threads are created, one of which fails to check for
     * a mutex lock leading to a null pointer dereference. In order to hit
     * the weakness qsize >= 5000000, benign values should use qsize <= 50.
     */
    private static ReentrantLock lock = new ReentrantLock();
    private static StringBuilder stonesoup_threadInput;

    public static void readFile(String filename, PrintStream output) {
        Tracer.tracepointLocation("STONESOUP_INJECTED_FILE", "readFile");
        String str;
        try {
            BufferedReader reader = new BufferedReader(new FileReader(filename));
            while ((str = reader.readLine()) != null) {
                output.println(str);
            }
            reader.close();
        } catch (FileNotFoundException e) {
            Tracer.tracepointError("Error reading syncFile.  " +
e.getClass().getName() + ": " + e.getMessage());
            output.println("Error reading sync file: " + e);
        } catch (IOException e) {
            Tracer.tracepointError("Error reading syncFile.  " +
e.getClass().getName() + ": " + e.getMessage());
            output.println("Error reading sync file: " + e);
        }
    }

    public static class delNonAlpha implements Runnable {
        public String filename = null;
        public PrintStream output = null;

        public void run() {
            Tracer.tracepointLocation("STONESOUP_INJECTED_FILE", "delNonAlpha.run");
            StringBuilder temp = new StringBuilder();
```

```
            try {
                for (int i = 0; i < stonesoup_threadInput.length(); i++) {
                    if(Character.isLetter(stonesoup_threadInput.charAt(i))) {
                        temp.append(stonesoup_threadInput.charAt(i));
                    }
                }

                Tracer.tracepointMessage("CROSSOVER-POINT: BEFORE");
                /* STONESOUP: CROSSOVER POINT (missing lock check) */
                stonesoup_threadInput = null;
                Tracer.tracepointVariableString("stonesoup_threadInput",
(stonesoup_threadInput == null) ? "(null)" : stonesoup_threadInput.toString());
                Tracer.tracepointMessage("CROSSOVER-POINT: AFTER");

                readFile(filename, output);
                stonesoup_threadInput = temp;
            }
            catch (java.lang.RuntimeException e) {
                e.printStackTrace(output);
                throw e;
            }
        }

        public delNonAlpha(String filename, PrintStream output) {
            Tracer.tracepointLocation("STONESOUP_INJECTED_FILE", "delNonAlpha.ctor");
            this.filename = filename;
            this.output = output;
        }
    }

    public static class toCap implements Runnable {

        public int size = 0;
        public String filename = null;
        public PrintStream output = null;

        public void run() {
            Tracer.tracepointLocation("STONESOUP_INJECTED_FILE", "toCap.run");
            try {
                int [] sortMe = new int[size];

                lock.lock();

                for (int i = 0; i < size; i++) {
                    sortMe[i] = size - i;
                }
                Arrays.sort(sortMe);
                readFile(filename, output);

                Tracer.tracepointMessage("TRIGGER-POINT: BEFORE");
                /* STONESOUP: TRIGGER POINT (missing lock check) */
                stonesoup_threadInput = new
StringBuilder().append(stonesoup_threadInput.toString().toUpperCase());
                Tracer.tracepointMessage("TRIGGER-POINT: AFTER");

                lock.unlock();
            }
            catch (java.lang.RuntimeException e) {
                e.printStackTrace(output);
```

```
                throw e;
            }
        }


        public toCap(int size, String filename, PrintStream output) {
            Tracer.tracepointLocation("STONESOUP_INJECTED_FILE", "toCap.ctor");
            this.size = size;
            this.filename = filename;
            this.output = output;
        }
    }

    public static void weakness(final String stonesoup_value, PrintStream
stonesoup_output) {
        Tracer.tracepointWeaknessStart("CWE414", "A", "Missing Lock Check");
        int stonesoup_qsize = 0;
        String stonesoup_taint = null;
        String stonesoup_file1 = null;
        String stonesoup_file2 = null;

        String stonesoup_substrings[] = stonesoup_value.split("\\s+", 4);

        if(stonesoup_substrings.length == 4){

            try{
                stonesoup_qsize = Integer.parseInt(stonesoup_substrings[0]);
                stonesoup_file1 = stonesoup_substrings[1];
                stonesoup_file2 = stonesoup_substrings[2];
                stonesoup_taint = stonesoup_substrings[3];

                Tracer.tracepointVariableString("stonesoup_value", stonesoup_value);
                Tracer.tracepointVariableInt("stonesoup_qsize", stonesoup_qsize);
                Tracer.tracepointVariableString("stonesoup_file1", stonesoup_file1);
                Tracer.tracepointVariableString("stonesoup_file2", stonesoup_file2);
                Tracer.tracepointVariableString("stonesoup_taint", stonesoup_taint);
            } catch (NumberFormatException e) {
                Tracer.tracepointError(e.getClass().getName() + ": " +
e.getMessage());
                stonesoup_output.println("NumberFormatException");
            }

            if (stonesoup_qsize < 0) {
                stonesoup_output.println("Error: use positive numbers.");
            } else {
                Tracer.tracepointMessage("Creating threads");

                Thread stonesoup_thread1 = new Thread(new toCap(stonesoup_qsize,
stonesoup_file1, stonesoup_output));
                Thread stonesoup_thread2 = new Thread(new delNonAlpha(stonesoup_file2,
stonesoup_output));

                stonesoup_threadInput = new StringBuilder().append(stonesoup_taint);

                stonesoup_output.println("Info: Spawning thread 1.");
                stonesoup_thread1.start();

                stonesoup_output.println("Info: Spawning thread 2.");
                stonesoup_thread2.start();
```

```
                try {
                    Tracer.tracepointMessage("Joining threads");
                    Tracer.tracepointMessage("Joining thread-01");
                    stonesoup_thread1.join();
                    Tracer.tracepointMessage("Joined thread-01");
                    Tracer.tracepointMessage("Joining thread-02");
                    stonesoup_thread2.join();
                    Tracer.tracepointMessage("Joined thread-02");
                    Tracer.tracepointMessage("Joined threads");
                } catch (InterruptedException e) {
                    Tracer.tracepointError(e.getClass().getName() + ": " +
e.getMessage());
                    stonesoup_output.println("Interrupted");
                }
                stonesoup_output.println("Info: Threads ended");
            }
        }
        Tracer.tracepointWeaknessEnd();
    }

    /**
     * @param args
     */
    public static void main(String[] args){
        if (args.length == 0) {
            throw new IllegalArgumentException(
                    "No arguments provided on command-line.");
        }
        String sourceTaint = args[0];
        if (sourceTaint == null) {
            throw new IllegalArgumentException(
                "No source taint provided in environment variable args[0].");
        }
        weakness(sourceTaint, System.err);
    }
```

```
}
```

J - CWE-460A Source Code

```
package com.pontetec.weakness.errorhandling;

import java.io.PrintStream;
import com.pontetec.stonesoup.trace.Tracer;

public class CWE460A {

    public static void weakness(final int size, PrintStream output) {
        /*
         * CWE-460 Improper Cleanup on Thrown Exception
         * This test takes input in the form of an integer
         * and creates an array of that size. If the input is
         * too large, an exception is raised and the array is
         * initialized to a default size without ever changing the
         * size variable. This leads to an ArrayIndexoutOfBoundsException
         * when the array is accessed.
         */
        Tracer.tracepointWeaknessStart("CWE460", "A", "Improper Cleanup on Thrown
Exception");

        int [] stonesoup_arr = null;

        Tracer.tracepointVariableInt("size", size);

        Tracer.tracepointMessage("CROSSOVER-POINT: BEFORE");
        try {
            output.printf("Allocating array of size %d\n", size);
            stonesoup_arr = new int[size];
        } catch (java.lang.OutOfMemoryError e) {
            Tracer.tracepointError(e.getClass().getName() + ": " + e.getMessage());
            /* STONESOUP: crossover point (improper cleanup) */
            stonesoup_arr = new int[100];
        }
        Tracer.tracepointBufferInfo("stonesoup_arr", stonesoup_arr.length, "Length of
stonesoup_arr");
        Tracer.tracepointMessage("CROSSOVER-POINT: AFTER");

        try {
            Tracer.tracepointMessage("TRIGGER-POINT: BEFORE");
            /* STONESOUP: trigger point (improper cleanup) */
            int i = size -1;
            do {
                // do stuff that includes incorrectly cleaned up data
                stonesoup_arr[i--] = i;
            } while (i > 0);
            Tracer.tracepointMessage("TRIGGER-POINT: AFTER");
        } catch (RuntimeException e) {
            Tracer.tracepointError(e.getClass().getName() + ": " + e.getMessage());
            e.printStackTrace(output);
            throw e;
        }
```

```
        Tracer.tracepointWeaknessEnd();
    }

    /**
     * @param args
     */
    public static void main(String[] args) {
        int value = 0;
        String sourceTaint = args[0];
        if (sourceTaint == null) {
                throw new IllegalArgumentException(
                                "No source taint provided in environment variable
args[0].");
        }
        try {
                value = Integer.parseInt(sourceTaint);
        } catch (NumberFormatException e) {
                throw new IllegalArgumentException(
                                "Source taint is not a valid short.", e);
        }
        weakness(value, System.err);
    }
}
```

J - CWE-543A Source Code

```
package com.pontetec.weakness.concurrency;

import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.io.PrintStream;
import java.math.BigInteger;
import java.util.Arrays;
import java.util.concurrent.LinkedBlockingQueue;
import com.pontetec.stonesoup.trace.Tracer;

public class CWE543A {
    /*
     * This weakness takes an integer, two file names, and an integer \"int file1
file2 int\"
     * as an input. The first integer is the size of the array to sort for timing in
benign cases
     * without FIFO files as inputs. The two files are used for thread \"scheduling\",
if they are
     * FIFO files in the order \"fifo1 fifo2\" with respect to the input script the
weakness will
     * occur, and if they are in reverse order the weakness will not occur. If the
files are given
     * as normal files, the weakness is dependent on the quicksort where a low
quicksort value is
     * nondeterministic and a larger value will increase the likelihood that the
weakness will not occur.
     * The final integer is the number of Fibonacci values to calculate and print.
This weakness uses
     * a singleton class to pass the Fibonacci values between the thread calculating
them and the
     * thread printing them. The method of instantiating a singleton used contains a
race condition
     * that can lead to multiple instances of the class being instantiated, leading to
deadlock.
     */

    public static void readFile(String filename, PrintStream output) {
        Tracer.tracepointLocation("STONESOUP_INJECTED_FILE", "readFile");
        String str;
        try {
            BufferedReader reader = new BufferedReader(new FileReader(filename));
            while ((str = reader.readLine()) != null) {
                output.println(str);
            }
            reader.close();
        } catch (FileNotFoundException e) {
            Tracer.tracepointError("Error reading syncFile.  " +
e.getClass().getName() + ": " + e.getMessage());
            output.println("Error reading sync file: " + e);
```

```
        } catch (IOException e) {
            Tracer.tracepointError("Error reading syncFile.  " +
e.getClass().getName() + ": " + e.getMessage());
            output.println("Error reading sync file: " + e);
        }
    }


    public static class JobHandler {                                // Singleton class
for passing data
        private LinkedBlockingQueue<BigInteger> data;                //    between
threads.
        private static JobHandler instance = null;

        private JobHandler() {
            // prevent instantiation outside of getInstance
            Tracer.tracepointLocation("STONESOUP_INJECTED_FILE", "JobHandler.ctor");
        }

        public static JobHandler getInstance(String filename, PrintStream output) {
            Tracer.tracepointLocation("STONESOUP_INJECTED_FILE",
"JobHandler.getInstance");
            if (instance == null) {
                Tracer.tracepointMessage("CROSSOVER-POINT: BEFORE");
                /* STONESOUP: CROSS-OVER POINT (Singleton without Sync) */

                readFile(filename, output);

                JobHandler temp = new JobHandler();
                temp.initialize();
                instance = temp;
                Tracer.tracepointMessage("CROSSOVER-POINT: AFTER");
                return temp;
            }
            return instance;
        }

        private void initialize() {
            Tracer.tracepointLocation("STONESOUP_INJECTED_FILE",
"JobHandler.initialize");
            data = new LinkedBlockingQueue<BigInteger>(30);
        }

        public void enqueue(BigInteger i) {
            Tracer.tracepointLocation("STONESOUP_INJECTED_FILE",
"JobHandler.enqueue");
            try {
                data.put(i);
            } catch (InterruptedException e) {
                throw new RuntimeException("Thread interrupted.", e);
            }
        }

        public BigInteger dequeue() {
            Tracer.tracepointLocation("STONESOUP_INJECTED_FILE",
"JobHandler.dequeue");
            try {
                return data.take();
            } catch (InterruptedException e) {
```

Approved for public release; distribution unlimited.

```
                throw new RuntimeException("Thread interrupted.", e);
            }
        }
    }

    public static class printData implements Runnable {
        private String filename;
        private PrintStream output;

        public void run() {
            Tracer.tracepointLocation("STONESOUP_INJECTED_FILE", "printData.run");
            JobHandler jobs = JobHandler.getInstance(filename, output);
            BigInteger i;

            Tracer.tracepointBuffer("printData: UID of JobHandler",
Integer.toHexString(System.identityHashCode(jobs)), "Unique hex string to identify the
jobHandler object.");
            Tracer.tracepointMessage("TRIGGER-POINT: BEFORE");
            /* STONESOUP: TRIGGER POINT (Singleton without Sync) */
            while ((i = jobs.dequeue()) != BigInteger.valueOf(-1)) {    // Read from
BlockingQueue, will wait indefinitely
                output.println(i.toString(10));                         //    if queue
is never populated.
            }
            Tracer.tracepointMessage("TRIGGER-POINT: AFTER");

        }

        public printData(String filename, PrintStream output) {
            Tracer.tracepointLocation("STONESOUP_INJECTED_FILE", "printData.ctor");
            this.filename = filename;
            this.output = output;
        }
    }

    public static class logData implements Runnable {
        private int size;
        private int numVal;
        private String filename;
        private PrintStream output;

        public void run() {
            Tracer.tracepointLocation("STONESOUP_INJECTED_FILE", "logData.run");

            int [] sortMe = new int[size];
            for (int i = 0; i < size; i++) {                          // Slow things
down for timing
                sortMe[i] = size - i;
            }
            Arrays.sort(sortMe);

            readFile(filename, output);

            JobHandler jobs = JobHandler.getInstance(filename, output);
            Tracer.tracepointBuffer("logData: UID of JobHandler",
Integer.toHexString(System.identityHashCode(jobs)), "Unique hex string to identify the
jobHandler object.");

            // populate data
```

```
            BigInteger a1 = BigInteger.valueOf(0);                    // Calculate
Fibonacci Sequence
            BigInteger a2 = BigInteger.valueOf(0);
            BigInteger c = BigInteger.valueOf(0);
            for (int i = 0; i < numVal; i++) {
                if (i == 0) {
                    jobs.enqueue(BigInteger.valueOf(0));
                }
                else if (i == 1) {
                    a1 = BigInteger.valueOf(1);
                    jobs.enqueue(BigInteger.valueOf(0));
                }
                else {
                    c = a1.add(a2);
                    a2 = a1;
                    a1 = c;
                    jobs.enqueue(c);
                }
            }
            jobs.enqueue(BigInteger.valueOf(-1));                        // Push data
to BlockingQueue, will wait indefinitely
        }                                                               //    if queue
fills up and is never emptied.

        public logData(int size, int numVal, String filename, PrintStream output) {
            Tracer.tracepointLocation("STONESOUP_INJECTED_FILE", "logData.ctor");
            this.numVal = numVal;
            this.size = size;
            this.filename = filename;
            this.output = output;
        }
    }

    public static void weakness(final String stonesoup_value, PrintStream
stonesoup_output) {
        Tracer.tracepointWeaknessStart("CWE543", "A", "Use of Singleton Pattern
Without Synchronization in a Multithreaded Context");

        int stonesoup_qsize = 0;
        int stonesoup_numVal = 0;
        String stonesoup_file1 = null;
        String stonesoup_file2 = null;

        String stonesoup_substrings[] = stonesoup_value.split("\\s+", 4);

        if(stonesoup_substrings.length == 4){

            try{
                stonesoup_qsize = Integer.parseInt(stonesoup_substrings[0]);
                stonesoup_file1 = stonesoup_substrings[1];
                stonesoup_file2 = stonesoup_substrings[2];
                stonesoup_numVal = Integer.parseInt(stonesoup_substrings[3]);

                Tracer.tracepointVariableString("stonesoup_value", stonesoup_value);
                Tracer.tracepointVariableInt("stonesoup_qsize", stonesoup_qsize);
                Tracer.tracepointVariableInt("stonesoup_numVal", stonesoup_numVal);
                Tracer.tracepointVariableString("stonesoup_file1", stonesoup_file1);
                Tracer.tracepointVariableString("stonesoup_file2", stonesoup_file2);
            }
```

Approved for public release; distribution unlimited.

```
            catch(NumberFormatException e){
                Tracer.tracepointError(e.getClass().getName() + ": " +
e.getMessage());
                stonesoup_output.println("NumberFormatException");
            }

            if (stonesoup_numVal <= 0 || stonesoup_qsize < 0) {
                stonesoup_output.println("Error: use positive numbers.");
            }
            else {
                Tracer.tracepointMessage("Creating threads");

                Thread stonesoup_thread1 = new Thread(new logData(stonesoup_qsize,
stonesoup_numVal, stonesoup_file1, stonesoup_output));
                Thread stonesoup_thread2 = new Thread(new printData(stonesoup_file2,
stonesoup_output));

                stonesoup_output.println("Info: Spawning thread 1.");
                stonesoup_thread1.start();

                stonesoup_output.println("Info: Spawning thread 2.");
                stonesoup_thread2.start();

                try {
                    Tracer.tracepointMessage("Joining threads");
                    Tracer.tracepointMessage("Joining thread-01");
                    stonesoup_thread1.join();
                    Tracer.tracepointMessage("Joined thread-01");
                    Tracer.tracepointMessage("Joining thread-02");
                    stonesoup_thread2.join();
                    Tracer.tracepointMessage("Joined thread-02");
                    Tracer.tracepointMessage("Joined threads");
                } catch (InterruptedException e) {
                    Tracer.tracepointError(e.getClass().getName() + ": " +
e.getMessage());
                    stonesoup_output.println("Interrupted");
                }
                stonesoup_output.println("Info: Threads ended");
            }
        }
    }

    /**
     * @param args
     */
    public static void main(String[] args){
        if (args.length == 0) {
            throw new IllegalArgumentException(
                    "No arguments provided on command-line.");
        }
        String sourceTaint = args[0];
        if (sourceTaint == null) {
            throw new IllegalArgumentException(
                "No source taint provided in environment variable args[0].");
        }
        weakness(sourceTaint, System.err);
    }
```

```
}
```

J - CWE-567A Source Code

```
package com.pontetec.weakness.concurrency;

import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.io.PrintStream;
import java.util.Arrays;
import com.pontetec.stonesoup.trace.Tracer;

public class CWE567A {
    /*
     * This weakness takes in an integer and string \"int string\" where the integer
     * is the size of the array to sort for timing and the string contains the value
that is acted
     * upon by the treads, stored in a global static variable. A divide by zero error
occurs when
     * the string starts with 'A' and the array size is significantly large. In
practice the benign
     * sort values are <50 and exploit are >5000000 to achieve (essentially)
guaranteed effects.
     */
    private static StringBuilder stonesoup_threadInput;
    private static volatile int dev_amount = 1;

    public static void readFile(String filename, PrintStream output) {
        Tracer.tracepointLocation("STONESOUP_INJECTED_FILE", "readFile");
        String str;
        try {
            BufferedReader reader = new BufferedReader(new FileReader(filename));
            while ((str = reader.readLine()) != null) {
                output.println(str);
            }
            reader.close();
        } catch (FileNotFoundException e) {
            Tracer.tracepointError("Error reading syncFile.  " +
e.getClass().getName() + ": " + e.getMessage());
            output.println("Error reading sync file: " + e);
        } catch (IOException e) {
            Tracer.tracepointError("Error reading syncFile.  " +
e.getClass().getName() + ": " + e.getMessage());
            output.println("Error reading sync file: " + e);
        }
    }

    public static class calcDevAmount implements Runnable {
        public String filename = null;
        public PrintStream output = null;

        public void run() {
            Tracer.tracepointLocation("STONESOUP_INJECTED_FILE", "calcDevAmount.run");
```

```
            try {
                Tracer.tracepointMessage("CROSSOVER-POINT: BEFORE");
                /* STONESOUP: CROSS-OVER POINT (unsynchronized access to shared data)
*/
                dev_amount = stonesoup_threadInput.charAt(0) - 'A';

                readFile(filename, output);
                Tracer.tracepointMessage("CROSSOVER-POINT: AFTER");

                if (dev_amount < 0) {
                    dev_amount *= -1;
                }
                if (dev_amount == 0) {
                    dev_amount += 1;
                }
            }
            catch (java.lang.RuntimeException e) {
                e.printStackTrace(output);
                throw e;
            }
        }

        public calcDevAmount(String filename, PrintStream output) {
            Tracer.tracepointLocation("STONESOUP_INJECTED_FILE",
"calcDevAmount.ctor");
            this.filename = filename;
            this.output = output;
        }
    }

    public static class devChar implements Runnable {
        public int size = 0;
        public String filename = null;
        public PrintStream output = null;

        public void run() {
            Tracer.tracepointLocation("STONESOUP_INJECTED_FILE", "devChar.run");
            int [] sortMe;
            sortMe = new int[size];

            try {
                for (int i = 0; i < size; i++) {
                    sortMe[i] = size - i;
                }
                Arrays.sort(sortMe);

                readFile(filename, output);

                Tracer.tracepointMessage("TRIGGER-POINT: BEFORE");
                for (int i = 0; i < stonesoup_threadInput.length(); i++) {
                    /* STONESOUP: TRIGGER POINT (unsynchronized access to shared data)
*/
                    stonesoup_threadInput.setCharAt(i,
(char)(stonesoup_threadInput.charAt(i) / dev_amount));
                }
                Tracer.tracepointMessage("TRIGGER-POINT: AFTER");

            }
```

```
            catch (java.lang.RuntimeException e) {
                e.printStackTrace(output);
                throw e;
            }
        }

        public devChar(int size, String filename, PrintStream output) {
            Tracer.tracepointLocation("STONESOUP_INJECTED_FILE", "devChar.ctor");
            this.size = size;
            this.filename = filename;
            this.output = output;
        }
    }

    public static void weakness(final String stonesoup_value, PrintStream
stonesoup_output) {
        Tracer.tracepointWeaknessStart("CWE567", "A", "Unsynchronized Access to Shared
Data in a Multithreaded Context");

        int stonesoup_qsize = 0;
        String stonesoup_taint = null;
        String stonesoup_file1 = null;
        String stonesoup_file2 = null;

        String stonesoup_substrings[] = stonesoup_value.split("\\s+", 4);

        if(stonesoup_substrings.length == 4){

            try{
                stonesoup_qsize = Integer.parseInt(stonesoup_substrings[0]);
                stonesoup_file1 = stonesoup_substrings[1];
                stonesoup_file2 = stonesoup_substrings[2];
                stonesoup_taint = stonesoup_substrings[3];

                Tracer.tracepointVariableString("stonesoup_value", stonesoup_value);
                Tracer.tracepointVariableInt("stonesoup_qsize", stonesoup_qsize);
                Tracer.tracepointVariableString("stonesoup_file1", stonesoup_file1);
                Tracer.tracepointVariableString("stonesoup_file2", stonesoup_file2);
                Tracer.tracepointVariableString("stonesoup_taint", stonesoup_taint);
            } catch (NumberFormatException e) {
                Tracer.tracepointError(e.getClass().getName() + ": " +
e.getMessage());
                stonesoup_output.println("NumberFormatException");
            }

            if (stonesoup_qsize < 0) {
                stonesoup_output.println("Error: use positive numbers.");
            } else {
                Tracer.tracepointMessage("Creating threads");

                Thread stonesoup_thread2 = new Thread(new devChar(stonesoup_qsize,
stonesoup_file1, stonesoup_output));
                Thread stonesoup_thread1 = new Thread(new
calcDevAmount(stonesoup_file2, stonesoup_output));

                stonesoup_threadInput = new StringBuilder().append(stonesoup_taint);

                stonesoup_output.println("Info: Spawning thread 1.");
                stonesoup_thread1.start();
```

Approved for public release; distribution unlimited.

```
                stonesoup_thread2.start();
                stonesoup_output.println("Info: Spawning thread 2.");

                try {
                    Tracer.tracepointMessage("Joining threads");
                    Tracer.tracepointMessage("Joining thread-01");
                    stonesoup_thread1.join();
                    Tracer.tracepointMessage("Joined thread-01");
                    Tracer.tracepointMessage("Joining thread-02");
                    stonesoup_thread2.join();
                    Tracer.tracepointMessage("Joined thread-02");
                    Tracer.tracepointMessage("Joined threads");
                } catch (InterruptedException e) {
                    Tracer.tracepointError(e.getClass().getName() + ": " +
e.getMessage());
                    stonesoup_output.println("Interrupted");
                }
                stonesoup_output.println("Info: Threads ended");
            }
        }
        Tracer.tracepointWeaknessEnd();
    }

    /**
     * @param args
     */
    public static void main(String[] args) {
        if (args.length == 0) {
            throw new IllegalArgumentException(
                    "No arguments provided on command-line.");
        }
        String sourceTaint = args[0];
        if (sourceTaint == null) {
            throw new IllegalArgumentException(
                    "No source taint provided in environment variable args[0].");
        }
        weakness(sourceTaint, System.err);
    }
```

```
}
```

J - CWE-572A Source Code

```
package com.pontetec.weakness.concurrency;

import java.io.BufferedInputStream;
import java.io.ByteArrayOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.PrintStream;
import java.nio.ByteBuffer;

import javax.xml.bind.DatatypeConverter;
import com.pontetec.stonesoup.trace.Tracer;

public class CWE572A {
    /*
     * A file name is provided to the application.   This file is assumed to be a
     * binary payload with at least 4 bytes of data.  The first 4 bytes of data
     * are an encoded 32-bit Integer.  This value describes the length of the
     * rest of the message.  If the length is not eual to of less than the
     * remainder of the message payload, a RuntimeException will be raised.
     *
     * Because the caller does not correctly use the Thread API to spawn a thread,
     * the main thread will receive the exception and not be handled accordingly.
     */
    public static class HelloRunnable implements Runnable {

        private PrintStream output;
        private String value;

        public void run() {
            Tracer.tracepointLocation("STONESOUP_INJECTED_FILE", "HelloRunnable.run");

            if (this.value == null) {
                return;
            }

            byte[] data = null;
            File filePath = new File("/opt/stonesoup/workspace/testData", this.value);
            BufferedInputStream inputStream = null;

            Tracer.tracepointVariableString("value", value);
            if (filePath.exists() && filePath.isFile()) {
                try {
                    /* STONESOUP: trigger point (read path is a name equivalent to a
rejected path) */
                    FileInputStream fis = new FileInputStream(filePath);

                    inputStream = new BufferedInputStream(fis);

                    byte[] inputBuffer = new byte[1024];
                    ByteArrayOutputStream byteArrayOutputStream = new
ByteArrayOutputStream();
```

```
                    int readAmount = 0;
                    while ((readAmount = inputStream.read(inputBuffer)) != -1) {
                        Tracer.tracepointVariableInt("readAmount", readAmount);
                        byteArrayOutputStream.write(inputBuffer, 0, readAmount);
                    }

                    data = byteArrayOutputStream.toByteArray();
                } catch (java.io.FileNotFoundException e) {
                    Tracer.tracepointError(e.getClass().getName() + ": " +
e.getMessage());
                    output.printf("File \"%s\" does not exist\n",
                            filePath.getPath());
                } catch (java.io.IOException ioe) {
                    Tracer.tracepointError(ioe.getClass().getName() + ": " +
ioe.getMessage());
                    output.println("Failed to read file.");
                } finally {
                    try {
                        if (inputStream != null) {
                            inputStream.close();
                        }
                    } catch (java.io.IOException e) {
                        output.println("STONESOUP: Closing file quietly.");
                    }
                }
            } else {
                output.printf("File \"%s\" does not exist\n",
                        filePath.getPath());
            }

            /* ensure that there are enough bytes to decode the length field */
            if (data == null || data.length < 4) {
                return;
            }

            /* a ByteBuffer provides more convenient access to decoding binary data
             * into Java types.
             */
            ByteBuffer buffer = ByteBuffer.wrap(data);
            int dataLength = buffer.getInt();

            /* ensure that data length is positive to avoid invalid reads */
            if (dataLength < 0) {
                return;
            } else if (dataLength == 0) {
                Tracer.tracepointError("Received payload with no data.");
                this.output.println("Received payload with no data.");
                return;
            }

            /* read the payload of the indictaed size */
            byte[] payload = new byte[dataLength];
            Tracer.tracepointBufferInfo("payload", payload.length, "Length of they
payload byte array.");
            Tracer.tracepointBufferInfo("buffer.position", buffer.position(),
"Position in buffer.");
            buffer.get(payload);
            Tracer.tracepointBufferInfo("buffer.position", buffer.position(),
```

Approved for public release; distribution unlimited.

```
"Position in buffer.");
            this.output.printf("Payload (Base64): %s\n",
DatatypeConverter.printBase64Binary(payload));
        }

        public HelloRunnable(String value, PrintStream output) {
            Tracer.tracepointLocation("STONESOUP_INJECTED_FILE",
"HelloRunnable.ctor");
            this.value = value;
            this.output = output;
        }
    }

    public static void weakness(final String stonesoup_value, PrintStream
stonesoup_output) {
        Tracer.tracepointWeaknessStart("CWE572", "A", "Call to Thread run() instead of
start()");

        Tracer.tracepointMessage("Creating thread");

        final PrintStream stonesoup_crash_output = stonesoup_output;
        Thread stonesoup_thread1 = new Thread(new HelloRunnable(stonesoup_value,
stonesoup_output));

        /* log the crash, but allow the main thread to continue */
        stonesoup_thread1.setUncaughtExceptionHandler(new
Thread.UncaughtExceptionHandler() {
            @Override
            public void uncaughtException(Thread t, Throwable e) {
                Tracer.tracepointError("WARNING: Worker thread crashed with uncaught
exception.");
                stonesoup_crash_output.println("WARNING: Worker thread crashed with
uncaught exception.");
                e.printStackTrace(stonesoup_crash_output);
            }
        });

        try {
            Tracer.tracepointMessage("CROSSOVER-POINT: BEFORE");
            Tracer.tracepointMessage("TRIGGER-POINT: BEFORE");
            /* STONESOUP: CROSS-OVER POINT (Call to Thread run instead of start) */
            /* STONESOUP: TRIGGER POINT (Call to Thread run instead of start) */
            stonesoup_thread1.run();
            Tracer.tracepointMessage("CROSSOVER-POINT: AFTER");
            Tracer.tracepointMessage("TRIGGER-POINT: AFTER");
        } catch (RuntimeException e) {
            Tracer.tracepointError("The thread startup raised an excpetion.  " +
e.getClass().getName() + ": " + e.getMessage());
            stonesoup_output.println("The thread startup raised an exception.  This
should never happen.");
            e.printStackTrace(stonesoup_output);
            throw e;
        }

        try {
            Tracer.tracepointMessage("Joining thread-01");
            stonesoup_thread1.join();
            Tracer.tracepointMessage("Joined thread-01");
        } catch (InterruptedException e1) {
```

```
            Tracer.tracepointError(e1.getClass().getName() + ": " + e1.getMessage());
            stonesoup_output.println("Failed to join the worker thread.");
            e1.printStackTrace(stonesoup_output);
        } finally {
            stonesoup_output.println("Worker thread terminated.");
        }
    }

    /**
     * @param args
     */
    public static void main(String[] args){
        if (args.length == 0) {
            throw new IllegalArgumentException(
                    "No arguments provided on command-line.");
        }
        String sourceTaint = args[0];
        if (sourceTaint == null) {
            throw new IllegalArgumentException(
                "No source taint provided in environment variable args[0].");
        }
        weakness(sourceTaint, System.err);
    }
```

```
}
```

J - CWE-584A Source Code

```
package com.pontetec.weakness.errorhandling;

import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintStream;
import java.util.Scanner;
import com.pontetec.stonesoup.trace.Tracer;

public class CWE584A {
    static class InvalidPathException extends Exception {
        private static final long serialVersionUID = 1L;

        public InvalidPathException(String msg) {
            super(msg);
        }
    }

    static class GetAbsolutePath {
        private String fileName;
        private PrintStream output;

        public GetAbsolutePath(String fileName, PrintStream output) {
            Tracer.tracepointLocation("STONESOUP_INJECTED_FILE",
"GetAbsolutePath.ctor");
            this.fileName = fileName;
            this.output = output;
        }

        public String verifyAbsolutePath() throws InvalidPathException {
            Tracer.tracepointLocation("STONESOUP_INJECTED_FILE",
"GetAbsolutePath.verifyAbsolutePath");

            String absName = null;
            File file = new File(fileName);
            if (file.exists()) {
                absName = file.getAbsolutePath();
            } else {
                throw (new InvalidPathException("No such file: " + fileName));
            }
            return absName;
        }

        @SuppressWarnings("finally")
        public String getAbsolutePath() throws InvalidPathException {
            Tracer.tracepointLocation("STONESOUP_INJECTED_FILE",
"GetAbsolutePath.getAbsolutePath");

            String absName = null;
            try {
                absName = this.verifyAbsolutePath();
            } catch (InvalidPathException e) {
```

```
                Tracer.tracepointError(e.getClass().getName() + ": " +
e.getMessage());
                output.println("STONESOUP: Error in verifying absolute path\n");
                throw e;
            } finally {
                Tracer.tracepointMessage("CROSSOVER-POINT: BEFORE");
                /* STONESOUP: CROSSOVER POINT (return inside finally) */
                return absName;
            }
        }
    }


    public static void weakness(final String taintedValue, PrintStream output) {
        /*
         * CWE-584 Return Inside Finally
         * This test takes the name of a file as input. It then attempts
         * to get the absolute path of the file and open it for reading.
         * Since GetAbsolutePath.getAbsolutePath has a return in a finally
         * clause, any generated warnings are dropped, eventually leading to
         * a null pointer exception.
         */
        Tracer.tracepointWeaknessStart("CWE584", "A", "Return Inside Finally");

        File file;
        Scanner freader;
        String absPath = null;
        GetAbsolutePath getpath = new GetAbsolutePath(taintedValue, output);
        boolean validPath = false;

        Tracer.tracepointVariableString("taintedValue", taintedValue);
        try {
            absPath = getpath.getAbsolutePath();
            Tracer.tracepointMessage("CROSSOVER-POINT: AFTER");
            validPath = true;
            Tracer.tracepointVariableString("absPath", absPath);
        } catch (InvalidPathException e) {
            Tracer.tracepointError(e.getClass().getName() + ": " + e.getMessage());
            output.println("STONESOUP: Absolute path to file was not found.");
        }

        if (validPath) {
            try {
                Tracer.tracepointMessage("TRIGGER-POINT: BEFORE");
                /* STONESOUP: TRIGGER POINT (return inside finally) */
                file = new File(absPath);
                freader = new Scanner(file);

                while (freader.hasNextLine()) {
                    output.println(freader.nextLine());
                }
                Tracer.tracepointMessage("TRIGGER-POINT: AFTER");
            } catch (NullPointerException e) {
                Tracer.tracepointError(e.getClass().getName() + ": " +
e.getMessage());
                e.printStackTrace(output);
                throw e;
            } catch (FileNotFoundException e) {
                Tracer.tracepointError(e.getClass().getName() + ": " +
```

```
e.getMessage());
                output.println("STONESOUP: File not found.");
            }
        }
        Tracer.tracepointWeaknessEnd();
    }

    /**
     * @param args
     * @throws Exception
     */
    public static void main(String[] args) {
        String sourceTaint = args[0];
        if (sourceTaint == null) {
                throw new IllegalArgumentException(
                                "No source taint provided in environment variable
args[0].");
        }
       weakness(sourceTaint, System.err);
    }
}
```

J - CWE-609A Source Code

```
package com.pontetec.weakness.concurrency;

import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.io.PrintStream;
import java.util.Arrays;
import java.util.concurrent.locks.ReentrantLock;
import com.pontetec.stonesoup.trace.Tracer;

public class CWE609A {
    /*
     * This weakness takes an input of "<qsize> <string>" where qsize is the length
     * of the array to generate and sort for timing and string is an arbitrary string
     * to use as data to pass around. The weakness uses a double checked lock to
initialize
     * a shared static data class in an attempt to be efficient (not block threads
when the
     * data is initialized) however this introduces a possibility for a thread to
return
     * from the method while another thread is in the process of initializing the
data.
     * This will lead to an access of uninitialized data, resulting in a
     * StringIndexOutOfBoundsException. This malicious behavior is triggered when
     * qsize >= 5000000, and not to occur when qsize <= 50.
     */

    private static ReentrantLock lock = new ReentrantLock();

    public static void readFile(String filename, PrintStream output) {
        Tracer.tracepointLocation("STONESOUP_INJECTED_FILE", "readFile");
        String str;
        try {
            BufferedReader reader = new BufferedReader(new FileReader(filename));
            while ((str = reader.readLine()) != null) {
                output.println(str);
            }
            reader.close();
        } catch (FileNotFoundException e) {
            Tracer.tracepointError("Error reading syncFile.  " +
e.getClass().getName() + ": " + e.getMessage());
            output.println("Error reading sync file: " + e);
        } catch (IOException e) {
            Tracer.tracepointError("Error reading syncFile.  " +
e.getClass().getName() + ": " + e.getMessage());
            output.println("Error reading sync file: " + e);
        }
    }

    public static class Stonesoup_Str {
```

```
        public static StringBuilder data = null;
        public static int size = -1;
    }

    public static void init_Stonesoup_Str(String data, int qsize, String filename,
PrintStream output){
        Tracer.tracepointLocation("STONESOUP_INJECTED_FILE", "init_Stonesoup_Str");
        output.println(Thread.currentThread().getId() + ": In init_Stonesoup_Str");
        if(Stonesoup_Str.data  == null){
            lock.lock();
            if(Stonesoup_Str.data == null){
                Tracer.tracepointMessage("CROSSOVER-POINT: BEFORE");
                /* STONESOUP: CROSS-OVER POINT (double checked lock) */
                Stonesoup_Str.data = new StringBuilder();
                Stonesoup_Str.size = data.length();

                output.println(Thread.currentThread().getId() + ": Initializing second
data");
                if (filename != null) {
                    readFile(filename, output);
                }

                Stonesoup_Str.data.append(data);
                Tracer.tracepointMessage("CROSSOVER-POINT: AFTER");
            } else {
                output.println(Thread.currentThread().getId() + ": No need to
initialize");
            }
            lock.unlock();
        } else {
            output.println(Thread.currentThread().getId() + ": Data is already
initialized");
        }
    }

    public static class doStuff implements Runnable {
        private int size = 0;
        private String data = null;
        private PrintStream output;
        String filename;

        public void run() {
            Tracer.tracepointLocation("STONESOUP_INJECTED_FILE", "doStuff.run");

            try {
                output.println(Thread.currentThread().getId() + ": Inside doStuff");

                Tracer.tracepointMessage("doStuff: entering init_Stonesoup_Str");
                init_Stonesoup_Str(data, size, filename, output);
                output.println(Thread.currentThread().getId() + ": In doStuff
Stonesoup_Str is: " + Stonesoup_Str.data.toString());

                Tracer.tracepointVariableString("Stonesoup_Str",
Stonesoup_Str.data.toString());
            }
            catch (java.lang.RuntimeException e) {
                e.printStackTrace(output);
                throw e;
            }
```

```
        }

        public doStuff(String data, int qsize, String filename, PrintStream output) {
            Tracer.tracepointLocation("STONESOUP_INJECTED_FILE", "doStuff.ctor");
            this.data = data;
            this.size = qsize;
            this.output = output;
            this.filename = filename;
        }
    }

    public static class doStuff2 implements Runnable {
        private int size = 0;
        private String data = null;
        private PrintStream output;
        private String filename;

        public void run() {
            Tracer.tracepointLocation("STONESOUP_INJECTED_FILE", "doStuff2.run");
            int[] sortMe = new int[size];

            try {
                output.println(Thread.currentThread().getId() + ": Inside doStuff2");
                for (int i = 0; i < size; i++){
                    sortMe[i] = size - i;
                }
                Arrays.sort(sortMe);

                readFile(filename, output);

                Tracer.tracepointMessage("doStuff2: entering init_Stonesoup_Str");
                init_Stonesoup_Str(data, size, null, output);

                for (int i = 0; i < Stonesoup_Str.data.length(); i++) {
                    if(Stonesoup_Str.data.charAt(i) >= 'a' ||
Stonesoup_Str.data.charAt(i) <= 'z'){
                        Stonesoup_Str.data.setCharAt(i, (char)
(Stonesoup_Str.data.charAt(i) - ('a' - 'A')));
                    }
                }

                Tracer.tracepointMessage("TRIGGER-POINT: BEFORE");
                /* STONESOUP: TRIGGER POINT (double checked lock) */
                if(Stonesoup_Str.data.charAt(0) != '\0'){
                    output.println(Thread.currentThread().getId() + ": In doStuff2
Stonesoup_Str is: " + Stonesoup_Str.data.toString());
                }
                Tracer.tracepointMessage("TRIGGER-POINT: AFTER");
            }
            catch (java.lang.RuntimeException e) {
                e.printStackTrace(output);
                throw e;
            }
        }

        public doStuff2(String data, int size, String filename, PrintStream output) {
            Tracer.tracepointLocation("STONESOUP_INJECTED_FILE", "doStuff2.ctor");
            this.data = data;
            this.size = size;
```

```
            this.filename = filename;
            this.output = output;
        }
    }

    public static void weakness(final String stonesoup_value, PrintStream
stonesoup_output) {
        Tracer.tracepointWeaknessStart("CWE609", "A", "Double-Checked Locking");

        int stonesoup_qsize = 0;
        String stonesoup_taint = null;
        String stonesoup_file1 = null;
        String stonesoup_file2 = null;

        String stonesoup_substrings[] = stonesoup_value.split("\\s+", 4);

        if(stonesoup_substrings.length == 4){

            try{
                stonesoup_qsize = Integer.parseInt(stonesoup_substrings[0]);
                stonesoup_file1 = stonesoup_substrings[1];
                stonesoup_file2 = stonesoup_substrings[2];
                stonesoup_taint = stonesoup_substrings[3];

                Tracer.tracepointVariableString("stonesoup_value", stonesoup_value);
                Tracer.tracepointVariableInt("stonesoup_qsize", stonesoup_qsize);
                Tracer.tracepointVariableString("stonesoup_file1", stonesoup_file1);
                Tracer.tracepointVariableString("stonesoup_file2", stonesoup_file2);
                Tracer.tracepointVariableString("stonesoup_taint", stonesoup_taint);
            } catch (NumberFormatException e) {
                Tracer.tracepointError(e.getClass().getName() + ": " +
e.getMessage());
                stonesoup_output.println("NumberFormatException");
            }

            if (stonesoup_qsize < 0) {
                stonesoup_output.println("Error: use positive numbers.");
            } else {
                Tracer.tracepointMessage("Creating threads");

                Thread stonesoup_thread1 = new Thread(new doStuff (stonesoup_taint,
stonesoup_qsize, stonesoup_file2, stonesoup_output));
                Thread stonesoup_thread2 = new Thread(new doStuff2(stonesoup_taint,
stonesoup_qsize, stonesoup_file1, stonesoup_output));

                stonesoup_output.println("Info: Spawning thread 1.");
                stonesoup_thread1.start();

                stonesoup_output.println("Info: Spawning thread 2.");
                stonesoup_thread2.start();

                try {
                    Tracer.tracepointMessage("Joining threads");
                    Tracer.tracepointMessage("Joining thread-01");
                    stonesoup_thread1.join();
                    Tracer.tracepointMessage("Joined thread-01");
                    Tracer.tracepointMessage("Joining thread-02");
                    stonesoup_thread2.join();
                    Tracer.tracepointMessage("Joined thread-02");
```

```
                    Tracer.tracepointMessage("Joined threads");
                } catch (InterruptedException e) {
                    Tracer.tracepointError(e.getClass().getName() + ": " +
e.getMessage());
                    stonesoup_output.println("Interrupted");
                }
                stonesoup_output.println("Info: Threads ended");
            }
        }
        Tracer.tracepointWeaknessEnd();
    }

    /**
     * @param args
     */
    public static void main(String[] args) {
        if (args.length == 0) {
            throw new IllegalArgumentException(
                    "No arguments provided on command-line.");
        }
        String sourceTaint = args[0];
        if (sourceTaint == null) {
            throw new IllegalArgumentException(
                    "No source taint provided in environment variable args[0].");
        }
        weakness(sourceTaint, System.err);
    }
```

```
}
```

J - CWE-663A Source Code

```
package com.pontetec.weakness.concurrency;

import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintStream;
import java.io.PrintWriter;
import java.util.Arrays;
import java.util.concurrent.locks.ReentrantLock;
import com.pontetec.stonesoup.trace.Tracer;

public class CWE663A {
    /*
     * This weakness takes an integer and string as input (int string) where the
integer
     * is the size of the array to sort for timing and the string is the data that is
acted
     * upon. The weakness spawns two threads, both of which in turn call the function
'arrFunc'
     * which is non-reentrant. 'arrFunc' uses a static integer to iterate through the
string
     * which can lead to multiple increments of j per iteration of the for loop
causing a
     * StringIndexOutOfBoundsException if 'arrFunc' is run simultaneously by two or
more threads.
     */
    private static ReentrantLock lock = new ReentrantLock();
    private static StringBuilder stonesoup_threadInput;
    static volatile int j;

    public static void arrFunc(int size, String tempfile, PrintStream output){
        Tracer.tracepointLocation("STONESOUP_INJECTED_FILE", "arrFunc");

        int[] sortMe = new int[size];
        j = 0;
        Tracer.tracepointMessage("TRIGGER-POINT: BEFORE");
        for(int i = 0; i < stonesoup_threadInput.length(); i++, j++){
            /* STONESOUP: TRIGGER POINT (non-reentrant function) */
            stonesoup_threadInput.setCharAt(j, '\0');                    // Increment
by static variable, if two threads are in the
                                                                        //    function
incrementing j, the variable will retain both
                                                                        //
increments, casing an StringIndexOutOfBoundsException
            output.format("TID: %d I: %d J: %d\n", Thread.currentThread().getId(), i,
j);

            if (size > 5) {
                try {

                    PrintWriter fileoutput = new PrintWriter(new BufferedWriter(new
```

```
FileWriter(tempfile)));
                    fileoutput.println("Iteration: " + i);
                    fileoutput.close();
                } catch (IOException e) {
                    Tracer.tracepointError("IOException");
                    // Do nothing
                }
                for (int k = 0; k < size; k++){                               //
Timing to ensure the weakness occurs
                    sortMe[k]=size-k;
                }
                Arrays.sort(sortMe);
            }
        }
        Tracer.tracepointMessage("TRIGGER-POINT: AFTER");
    }

    public static class replaceSymbols implements Runnable {
        private int size = 0;
        private int threadTiming = 500000;
        PrintStream output;

        public void run() {
            Tracer.tracepointLocation("STONESOUP_INJECTED_FILE",
"replaceSymbols.run");

            try {
                int [] sortMe = new int[threadTiming];                    // Timing to
prevent the weakness on good cases
                for (int k = 0; k < threadTiming; k++) {
                    sortMe[k] = threadTiming - k;
                }
                Arrays.sort(sortMe);

                Tracer.tracepointMessage("replaceSymbols: after qsort");

                lock.lock();
                char val;
                for(int i = 0; i < stonesoup_threadInput.length(); i++){
                    val=stonesoup_threadInput.charAt(i);
                    if(((val >= '!' && val <= '/') ||
                        (val >= ':' && val <= '@') ||
                        (val >= '[' && val <= '`') ||
                        (val >= '{' && val <= '~')) &&
                        (val != '@' && val != '.')) {
                            stonesoup_threadInput.setCharAt(i, '_');
                    }
                }

                lock.unlock();

                Tracer.tracepointMessage("CROSSOVER-POINT: BEFORE (1)");
                /* STONESOUP: CROSS-OVER POINT (non-reentrant function) */
                arrFunc(size, "/opt/stonesoup/workspace/testData/replace.txt",
output);
                Tracer.tracepointMessage("CROSSOVER-POINT: AFTER (1)");
            }
            catch (java.lang.RuntimeException e) {
                e.printStackTrace(output);
```

```
                throw e;
            }
        }

        public replaceSymbols(int size, PrintStream output) {
            Tracer.tracepointLocation("STONESOUP_INJECTED_FILE",
"replaceSymbols.ctor");
            this.size = size;
            this.output = output;
        }
    }

    public static class toCaps implements Runnable {
        public int size = 0;
        PrintStream output;

        public void run() {
            Tracer.tracepointLocation("STONESOUP_INJECTED_FILE", "toCaps.run");

            try {
                lock.lock();

                for (int i = 0; i < stonesoup_threadInput.length(); i++) {
                    if(stonesoup_threadInput.charAt(i) >= 'a' ||
stonesoup_threadInput.charAt(i) <= 'z'){
                        stonesoup_threadInput.setCharAt(i, (char)
(stonesoup_threadInput.charAt(i) - ('a' - 'A')));
                    }
                }
                lock.unlock();

                Tracer.tracepointMessage("CROSSOVER-POINT: BEFORE (2)");
                /* STONESOUP2: CROSS-OVER POINT (non-reentrant function) */
                arrFunc(size, "/opt/stonesoup/workspace/testData/toCaps.txt", output);
                Tracer.tracepointMessage("CROSSOVER-POINT: AFTER (2)");
            }
            catch (java.lang.RuntimeException e) {
                e.printStackTrace(output);
                throw e;
            }
        }

        public toCaps(int size, PrintStream output) {
            Tracer.tracepointLocation("STONESOUP_INJECTED_FILE", "toCaps.ctor");
            this.size = size;
            this.output = output;
        }
    }

    public static void weakness(final String stonesoup_value, PrintStream
stonesoup_output) {
        Tracer.tracepointWeaknessStart("CWE663", "A", "Use of a Non-reentrant Function
in a Concurrent Context");

        String stonesoup_substrings[] = stonesoup_value.split("\\s",2);
        int stonesoup_qsize = 0;
        if (stonesoup_substrings.length == 2) {

            try {
```

```
                stonesoup_qsize = Integer.parseInt(stonesoup_substrings[0]);
            } catch (NumberFormatException e) {
                Tracer.tracepointError(e.getClass().getName() + ": " +
e.getMessage());
                stonesoup_output.println("NumberFormatException");
            }

            Tracer.tracepointVariableString("stonesoup_value", stonesoup_value);
            Tracer.tracepointVariableInt("stonesoup_qsize", stonesoup_qsize);
            Tracer.tracepointVariableString("stonesoup_threadInput",
stonesoup_substrings[1]);

            if (stonesoup_qsize < 0) {
                stonesoup_qsize = 0;
                stonesoup_output.println("Qsize should be >=0, setting it to 0.");
            }

            Tracer.tracepointVariableInt("stonesoup_qsize", stonesoup_qsize);
            Tracer.tracepointMessage("Creating threads");

            Thread stonesoup_thread1 = new Thread(new replaceSymbols(stonesoup_qsize,
stonesoup_output));
            Thread stonesoup_thread2 = new Thread(new toCaps(stonesoup_qsize,
stonesoup_output));

            stonesoup_threadInput = new
StringBuilder().append(stonesoup_substrings[1]);

            Tracer.tracepointMessage("Spawning threads.");
            stonesoup_output.println("Info: Spawning thread 1.");
            stonesoup_thread1.start();

            stonesoup_output.println("Info: Spawning thread 2.");
            stonesoup_thread2.start();

            try {
                Tracer.tracepointMessage("Joining threads");
                Tracer.tracepointMessage("Joining thread-01");
                stonesoup_thread1.join();
                Tracer.tracepointMessage("Joined thread-01");
                Tracer.tracepointMessage("Joining thread-02");
                stonesoup_thread2.join();
                Tracer.tracepointMessage("Joined thread-02");
                Tracer.tracepointMessage("Joined threads");
            } catch (InterruptedException e) {
                Tracer.tracepointError(e.getClass().getName() + ": " +
e.getMessage());
                stonesoup_output.println("Interrupted");
            }
            stonesoup_output.println("Info: Threads ended");
        }
        Tracer.tracepointWeaknessEnd();
    }

    /**
     * @param args
     */
    public static void main(String[] args) {
        if (args.length == 0) {
```

```
            throw new IllegalArgumentException(
                    "No arguments provided on command-line.");
        }
        String sourceTaint = args[0];
        if (sourceTaint == null) {
            throw new IllegalArgumentException(
                    "No source taint provided in environment variable args[0].");
        }
        weakness(sourceTaint, System.err);
    }
```

```
}
```

J - CWE-764A Source Code

```
package com.pontetec.weakness.concurrency;

import java.io.PrintStream;
import java.util.concurrent.locks.ReentrantLock;
import com.pontetec.stonesoup.trace.Tracer;

public class CWE764A {
    /*
     * This test takes a string.  It locks a lock upon encountering the first
     * '1' in the string, then locks the lock each time it subsequently
     * encounters another '1'.    If there are three or more '1' characters in
     * the string, this will cause multiple locks and an unexpected state (DOS).
     */
    public static class HelloRunnable implements Runnable {
            private static ReentrantLock lock;
            private static int count;

            private String input;
            private PrintStream output;

            public int getCount() {
                return count;
            }

            public void run() {
                Tracer.tracepointLocation("STONESOUP_INJECTED_FILE",
"HelloRunnable.run");
                int index = 0;
                while (index < input.length()) {
                    char cc = input.charAt(index);
                    index++;
                    if (cc == '1') {
                        lock.lock();
                        break;
                    }
                }

                boolean found1 = false;
                while (index < input.length()) {
                    char cc = input.charAt(index);
                    index++;
                    if (!found1) {
                        count++;
                    }
                    if (cc == '1') {
                        /* STONESOUP: CROSS-OVER POINT (Multiple locks of a critical
resource) */
                        /* STONESOUP: TRIGGER POINT (Multiple locks of a critical
resource) */
                        lock.lock();
                        found1 = true;
```

```
                    }
                }
                if (lock.isHeldByCurrentThread()) {
                    if (lock.getHoldCount() > 1) {
                        lock.unlock();
                    }
                    lock.unlock();
                }
                output.println(
                        "Info: Found " + getCount() + " letters between 1 and 1");
            }


            public HelloRunnable(String input, PrintStream output) {
                Tracer.tracepointLocation("STONESOUP_INJECTED_FILE",
"HelloRunnable.ctor");
                lock = new ReentrantLock();
                count = 0;
                this.input = input;
                this.output = output;
            }
        }

        public static void weakness(final String stonesoup_value, PrintStream
stonesoup_output) {
            Tracer.tracepointWeaknessStart("CWE764", "A", "Multiple Locks of a
Critical Resource");

            Tracer.tracepointVariableString("stonesoup_value", stonesoup_value);

            Tracer.tracepointMessage("Creating threads");

            Thread stonesoup_thread1 = new Thread(new HelloRunnable(stonesoup_value,
stonesoup_output));
            Thread stonesoup_thread2 = new Thread(new HelloRunnable(stonesoup_value,
stonesoup_output));

            Tracer.tracepointMessage("CROSSOVER-POINT: BEFORE");
            Tracer.tracepointMessage("TRIGGER-POINT: BEFORE");

            stonesoup_thread1.start();
            stonesoup_thread2.start();

            try {
                Tracer.tracepointMessage("Joining threads");
                Tracer.tracepointMessage("Joining thread-01");
                stonesoup_thread1.join();
                Tracer.tracepointMessage("Joined thread-01");
                Tracer.tracepointMessage("Joining thread-02");
                stonesoup_thread2.join();
                Tracer.tracepointMessage("Joined thread-02");
                Tracer.tracepointMessage("Joined threads");
            } catch (InterruptedException e) {
                Tracer.tracepointError(e.getClass().getName() + ": " +
e.getMessage());
                stonesoup_output.println("Interrupted");
            }
            Tracer.tracepointMessage("TRIGGER-POINT: AFTER");
            Tracer.tracepointMessage("CROSSOVER-POINT: AFTER");
```

```
        stonesoup_output.println("Info: Threads ended");
        Tracer.tracepointWeaknessEnd();
    }

    /**
     * @param args
     */
    public static void main(String[] args){
        if (args.length == 0) {
            throw new IllegalArgumentException(
                    "No arguments provided on command-line.");
        }
        String sourceTaint = args[0];
        if (sourceTaint == null) {
            throw new IllegalArgumentException(
                "No source taint provided in environment variable args[0].");
        }
        weakness(sourceTaint, System.err);
    }
```

```
}
```

J - CWE-765A Source Code

```
package com.pontetec.weakness.concurrency;

import java.io.PrintStream;
import java.util.concurrent.locks.ReentrantLock;
import com.pontetec.stonesoup.trace.Tracer;

public class CWE765A {
    /*
     * This test takes a string.  It locks a lock upon encountering the first
     * '1' in the string, then unlocks the lock each time it subsequently
     * encounters an '1'.  If there are three or more '1' characters in the
     * string, this will cause multiple unlocks and an exception.
     */
    public static class HelloRunnable implements Runnable {
        private static ReentrantLock lock;
        private static int count;

        private String input;
        private PrintStream output;

        public int getCount() {
            return count;
        }

        public void run() {
            Tracer.tracepointLocation("STONESOUP_INJECTED_FILE", "HelloRunnable.run");
            Tracer.tracepointVariableString("input", input);

            try {
                int index = 0;
                while (index < input.length()) {
                    char cc = input.charAt(index);
                    index ++;
                    if (cc == '1') {
                        Tracer.tracepointMessage("Locking lock");
                        Tracer.tracepointVariableInt("index", index);
                        lock.lock();
                        break;
                    }
                }

                Tracer.tracepointMessage("CROSSOVER-POINT: BEFORE");
                Tracer.tracepointMessage("TRIGGER-POINT: BEFORE");
                boolean found1 = false;
                while (index < input.length()) {
                    char cc = input.charAt(index);
                    index ++;
                    if (!found1) {
                        count++;
                    }
```

```
                    if (cc == '1') {
                        /* STONESOUP: CROSS-OVER POINT (Multiple unlocks of a critical
resource) */
                        /* STONESOUP: TRIGGER POINT (Multiple unlocks of a critical
resource) */
                        Tracer.tracepointMessage("Unlocking lock");
                        lock.unlock();
                        found1 = true;
                    }
                }
                if (lock.isHeldByCurrentThread()) {
                    Tracer.tracepointMessage("Unlocking lock");
                    lock.unlock();
                }
                Tracer.tracepointMessage("TRIGGER-POINT: AFTER");
                Tracer.tracepointMessage("CROSSOVER-POINT: AFTER");
                output.println(
                        "Info: Found " + getCount() + " letters between 1 and 1");
            }
            catch (java.lang.RuntimeException e) {
                e.printStackTrace(output);
                throw e;
            }
        }


        public HelloRunnable(String input, PrintStream output) {
            Tracer.tracepointLocation("STONESOUP_INJECTED_FILE",
"HelloRunnable.ctor");
            lock = new ReentrantLock();
            count = 0;
            this.input = input;
            this.output = output;
        }
    }

    public static void weakness(final String stonesoup_value, PrintStream
stonesoup_output) {
        Tracer.tracepointWeaknessStart("CWE765", "A", "Multiple Unlocks of a Critical
Resource");

        Tracer.tracepointMessage("Creating thread");

        Thread stonesoup_thread1 = new Thread(new HelloRunnable(stonesoup_value,
stonesoup_output));
        stonesoup_thread1.start();

        try {
            Tracer.tracepointMessage("Joining thread-01");
            stonesoup_thread1.join();
            Tracer.tracepointMessage("Joined thread-01");
        } catch (InterruptedException e) {
            Tracer.tracepointError(e.getClass().getName() + ": " + e.getMessage());
            stonesoup_output.println("Interrupted");
        }
        stonesoup_output.println("Info: Threads ended");

        Tracer.tracepointWeaknessEnd();
    }
```

```
/**
 * @param args
 */
public static void main(String[] args){
    if (args.length == 0) {
        throw new IllegalArgumentException(
                "No arguments provided on command-line.");
    }
    String sourceTaint = args[0];
    if (sourceTaint == null) {
        throw new IllegalArgumentException(
            "No source taint provided in environment variable args[0].");
    }
    weakness(sourceTaint, System.err);
}
```

```
}
```

J - CWE-820A Source Code

```
package com.pontetec.weakness.concurrency;

import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.io.PrintStream;
import java.util.Arrays;
import com.pontetec.stonesoup.trace.Tracer;

public class CWE820A {
    /*
     * This test takes a string.  It separates the string into a number to use
     * for an array size to sort and a second string to share between threads.
     * It then creates two threads and runs them both.    The first thread calculates
     * an increment amount that is used in the second thread.  If the string contains
     * a number or lower decimal value ascii character it will cause the increment
amount
     * to become negative which won't be fixed before the second thread uses it
causing
     * a StringIndexOutOfBoundsException.
     */

    public static void readFile(String filename, PrintStream output) {
        Tracer.tracepointLocation("STONESOUP_INJECTED_FILE", "readFile");
        String str;
        try {
            BufferedReader reader = new BufferedReader(new FileReader(filename));
            while ((str = reader.readLine()) != null) {
                output.println(str);
            }
            reader.close();
        } catch (FileNotFoundException e) {
            Tracer.tracepointError("Error reading syncFile.  " +
e.getClass().getName() + ": " + e.getMessage());
            output.println("Error reading sync file: " + e);
        } catch (IOException e) {
            Tracer.tracepointError("Error reading syncFile.  " +
e.getClass().getName() + ": " + e.getMessage());
            output.println("Error reading sync file: " + e);
        }
    }

    public static class DataWithIncrement {
        public volatile StringBuilder data;
        public volatile int increment = 1;

        public DataWithIncrement(int increment, StringBuilder data) {
            Tracer.tracepointLocation("STONESOUP_INJECTED_FILE",
"DataWithIncrement.ctor");
            this.increment = increment;
```

```
            this.data = data;
        }
    }

    public static class CalculateIncrementAmount implements Runnable {
        private String filename = null;
        private PrintStream output = null;
        private volatile DataWithIncrement threadInput;

        public void run() {
            Tracer.tracepointLocation("STONESOUP_INJECTED_FILE",
"CalculateIncrementAmount.run");

            try {
                Tracer.tracepointMessage("CROSSOVER-POINT: BEFORE");
                /* STONESOUP: CROSS-OVER POINT (missing sync) */
                threadInput.increment = threadInput.data.charAt(0) - 'A';
                Tracer.tracepointVariableInt("threadInput.increment",
threadInput.increment);
                Tracer.tracepointMessage("CROSSOVER-POINT: AFTER");

                readFile(filename, output);

                if (this.threadInput.increment < 0) {
                    this.threadInput.increment *= -1;
                } else if (this.threadInput.increment == 0) {
                    this.threadInput.increment += 1;
                }
                Tracer.tracepointVariableInt("threadInput.increment",
threadInput.increment);
            }
            catch (java.lang.RuntimeException e) {
                e.printStackTrace(output);
                throw e;
            }
        }

        public CalculateIncrementAmount(DataWithIncrement input, String filename,
PrintStream output) {
            Tracer.tracepointLocation("STONESOUP_INJECTED_FILE",
"CalculateIncrementAmount.ctor");
            this.threadInput = input;
            this.filename = filename;
            this.output = output;
        }
    }

    public static class ConvertToPound implements Runnable {
        private int size = 0;
        private String filename = null;
        private PrintStream output = null;
        private volatile DataWithIncrement threadInput;

        public void run() {
            Tracer.tracepointLocation("STONESOUP_INJECTED_FILE",
"ConvertToPound.run");
            int[] sortMe = new int[size];

            try {
```

```
                for (int i = 0; i < this.size; i++) {
                    sortMe[i] = this.size - i;
                }
                Arrays.sort(sortMe);

                readFile(filename, output);

                Tracer.tracepointMessage("TRIGGER-POINT: BEFORE");
                Tracer.tracepointVariableInt("threadInput.increment",
threadInput.increment);
                /* STONESOUP: TRIGGER POINT (missing sync) */
                for (int i = 0; i < this.threadInput.data.length(); i +=
this.threadInput.increment) {
                    this.threadInput.data.setCharAt(i, '#');
                }
                Tracer.tracepointMessage("TRIGGER-POINT: AFTER");
            }
            catch (java.lang.RuntimeException e) {
                e.printStackTrace(output);
                throw e;
            }
        }


        public ConvertToPound(int size, DataWithIncrement input, String filename,
PrintStream output) {
            Tracer.tracepointLocation("STONESOUP_INJECTED_FILE",
"ConvertToPound.ctor");
            this.size = size;
            this.threadInput = input;
            this.filename = filename;
            this.output = output;
        }
    }

    public static void weakness(final String stonesoup_value, PrintStream
stonesoup_output) {
        Tracer.tracepointWeaknessStart("CWE820", "A", "Missing Synchronization");

        int stonesoup_qsize = 0;
        String stonesoup_taint = null;
        String stonesoup_file1 = null;
        String stonesoup_file2 = null;

        String stonesoup_substrings[] = stonesoup_value.split("\\s+", 4);

        if(stonesoup_substrings.length == 4){

            try{
                stonesoup_qsize = Integer.parseInt(stonesoup_substrings[0]);
                stonesoup_file1 = stonesoup_substrings[1];
                stonesoup_file2 = stonesoup_substrings[2];
                stonesoup_taint = stonesoup_substrings[3];

                Tracer.tracepointVariableString("stonesoup_value", stonesoup_value);
                Tracer.tracepointVariableInt("stonesoup_qsize", stonesoup_qsize);
                Tracer.tracepointVariableString("stonesoup_file1", stonesoup_file1);
                Tracer.tracepointVariableString("stonesoup_file2", stonesoup_file2);
                Tracer.tracepointVariableString("stonesoup_taint", stonesoup_taint);
```

```
            } catch (NumberFormatException e) {
                Tracer.tracepointError(e.getClass().getName() + ": " +
e.getMessage());
                stonesoup_output.println("NumberFormatException");
            }

            if (stonesoup_qsize < 0) {
                stonesoup_output.println("Error: use positive numbers.");
            } else {

                DataWithIncrement stonesoup_input_data = new DataWithIncrement(0, new
StringBuilder().append(stonesoup_taint));

                Tracer.tracepointMessage("Creating threads");

                Thread stonesoup_thread1 = new Thread(new
CalculateIncrementAmount(stonesoup_input_data, stonesoup_file2, stonesoup_output));
                Thread stonesoupthread2 = new Thread(new
ConvertToPound(stonesoup_qsize, stonesoup_input_data, stonesoup_file1,
stonesoup_output));

                stonesoup_output.println("Info: Spawning thread 1.");
                stonesoup_thread1.start();

                stonesoup_output.println("Info: Spawning thread 2.");
                stonesoupthread2.start();

                try {
                    Tracer.tracepointMessage("Joining threads");
                    Tracer.tracepointMessage("Joining thread-01");
                    stonesoup_thread1.join();
                    Tracer.tracepointMessage("Joined thread-01");
                    Tracer.tracepointMessage("Joining thread-02");
                    stonesoupthread2.join();
                    Tracer.tracepointMessage("Joined thread-02");
                    Tracer.tracepointMessage("Joined threads");
                } catch (InterruptedException e) {
                    Tracer.tracepointError(e.getClass().getName() + ": " +
e.getMessage());
                    stonesoup_output.println("Interrupted");
                }
                stonesoup_output.println("Info: Threads ended");

                Tracer.tracepointWeaknessEnd();
            }
        }
    }

    /**
     * @param args
     */
    public static void main(String[] args) {
        if (args.length == 0) {
            throw new IllegalArgumentException(
                    "No arguments provided on command-line.");
        }
        String sourceTaint = args[0];
        if (sourceTaint == null) {
            throw new IllegalArgumentException(
```

```
                "No source taint provided in environment variable args[0].");
        }
        weakness(sourceTaint, System.err);
    }
```

Approved for public release; distribution unlimited.

```
}
```

J - CWE-821A Source Code

```
package com.pontetec.weakness.concurrency;

import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.io.PrintStream;
import java.util.Arrays;
import java.util.concurrent.locks.ReentrantLock;

import com.pontetec.stonesoup.trace.Tracer;

public class CWE821A {
    /*
     * This weakness takes in an integer and string \"int string\" where the integer
     * is the size of the array to sort for timing and the string contains the value
that is acted
     * upon by the treads, passed as a global static class. A divide by zero error
occurs when
     * the string starts with 'A' and the array size is significantly large. In
practice the benign
     * sort values are <50 and exploit are >5000000 to achieve (essentially)
guaranteed effects.
     */
    private static ReentrantLock lock = new ReentrantLock();
    private static ReentrantLock lock2 = new ReentrantLock();
    private static StringBuilder stonesoup_threadInput;

    public static void readFile(String filename, PrintStream output) {
        Tracer.tracepointLocation("STONESOUP_INJECTED_FILE", "readFile");
        String str;
        try {
            BufferedReader reader = new BufferedReader(new FileReader(filename));
            while ((str = reader.readLine()) != null) {
                output.println(str);
            }
            reader.close();
        } catch (FileNotFoundException e) {
            Tracer.tracepointError("Error reading syncFile.  " +
e.getClass().getName() + ": " + e.getMessage());
            output.println("Error reading sync file: " + e);
        } catch (IOException e) {
            Tracer.tracepointError("Error reading syncFile.  " +
e.getClass().getName() + ": " + e.getMessage());
            output.println("Error reading sync file: " + e);
        }
    }

    public static class Stonesoup_Int {
        int i;
        public Stonesoup_Int(int i) {
```

```
            this.i = i;
        }

        public int getVal() {
            return i;
        }

        public void setVal(int i) {
            this.i = i;
        }
    }

    public static class calcDevAmount implements Runnable {
        private Stonesoup_Int dev_amount;
        private String filename = null;
        private PrintStream output = null;

        public void run() {
            Tracer.tracepointLocation("STONESOUP_INJECTED_FILE", "calcDevAmount.run");

            try {
                lock.lock();

                Tracer.tracepointMessage("CROSSOVER-POINT: BEFORE");
                /* STONESOUP: CROSS-OVER POINT (Incorrect Synchronization) */
                dev_amount.setVal(stonesoup_threadInput.charAt(0) - 'A');
                Tracer.tracepointVariableInt("dev_amount.getVal()",
dev_amount.getVal());
                Tracer.tracepointMessage("CROSSOVER-POINT: AFTER");

                readFile(filename, output);

                if (dev_amount.getVal() < 0) {
                    dev_amount.setVal(dev_amount.getVal() * -1);
                }
                if (dev_amount.getVal() == 0) {
                    dev_amount.setVal(dev_amount.getVal() + 1);
                }

                Tracer.tracepointVariableInt("dev_amount.getVal()",
dev_amount.getVal());
                lock.unlock();
            }
            catch (java.lang.RuntimeException e) {
                e.printStackTrace(output);
                throw e;
            }
        }

        public calcDevAmount(Stonesoup_Int dev_amount, String filename, PrintStream
output) {
            Tracer.tracepointLocation("STONESOUP_INJECTED_FILE",
"calcDevAmount.ctor");
            this.dev_amount = dev_amount;
            this.filename = filename;
            this.output = output;
        }
    }
```

```
    public static class devChar implements Runnable {
        private int size = 0;
        private Stonesoup_Int dev_amount;
        private String filename = null;
        private PrintStream output = null;

        public void run() {
            Tracer.tracepointLocation("STONESOUP_INJECTED_FILE", "devChar.run");

            try {
                lock2.lock();
                int[] sortMe = new int[size];

                for (int i = 0; i < size; i++) {
                    sortMe[i] = size - i;
                }
                Arrays.sort(sortMe);

                readFile(filename, output);

                Tracer.tracepointMessage("TRIGGER-POINT: BEFORE");
                Tracer.tracepointVariableInt("dev_amount.getVal()",
dev_amount.getVal());
                for (int i = 0; i < stonesoup_threadInput.length(); i++) {
                    /* STONESOUP: TRIGGER POINT (Incorrect Synchronization) */
                    stonesoup_threadInput.setCharAt(i,
(char)(stonesoup_threadInput.charAt(i) / dev_amount.getVal()));
                }
                Tracer.tracepointMessage("TRIGGER-POINT: AFTER");
                lock2.unlock();
            }
            catch (java.lang.RuntimeException e) {
                e.printStackTrace(output);
                throw e;
            }
        }

        public devChar(int size, Stonesoup_Int dev_amount, String filename,
PrintStream output) {
            Tracer.tracepointLocation("STONESOUP_INJECTED_FILE", "devChar.ctor");
            this.size = size;
            this.dev_amount = dev_amount;
            this.filename = filename;
            this.output = output;
        }
    }

    public static void weakness(final String stonesoup_value, PrintStream
stonesoup_output) {
        Tracer.tracepointWeaknessStart("CWE821", "A", "Incorrect Synchronization");

        Stonesoup_Int stonesoup_dev_amount = new Stonesoup_Int(1);
        int stonesoup_qsize = 0;
        String stonesoup_taint = null;
        String stonesoup_file1 = null;
        String stonesoup_file2 = null;

        String stonesoup_substrings[] = stonesoup_value.split("\\s+", 4);
```

```
        if(stonesoup_substrings.length == 4){

            try{
                stonesoup_qsize = Integer.parseInt(stonesoup_substrings[0]);
                stonesoup_file1 = stonesoup_substrings[1];
                stonesoup_file2 = stonesoup_substrings[2];
                stonesoup_taint = stonesoup_substrings[3];

                Tracer.tracepointVariableString("stonesoup_value", stonesoup_value);
                Tracer.tracepointVariableInt("stonesoup_qsize", stonesoup_qsize);
                Tracer.tracepointVariableString("stonesoup_file1", stonesoup_file1);
                Tracer.tracepointVariableString("stonesoup_file2", stonesoup_file2);
                Tracer.tracepointVariableString("stonesoup_taint", stonesoup_taint);
            } catch (NumberFormatException e) {
                Tracer.tracepointError(e.getClass().getName() + ": " +
e.getMessage());
                stonesoup_output.println("NumberFormatException");
            }

            if (stonesoup_qsize < 0) {
                stonesoup_output.println("Error: use positive numbers.");
            } else {

                Tracer.tracepointMessage("Creating threads");

                Thread stonesoup_thread2 = new Thread(new devChar(stonesoup_qsize,
stonesoup_dev_amount, stonesoup_file1, stonesoup_output));
                Thread stonesoup_thread1 = new Thread(new
calcDevAmount(stonesoup_dev_amount, stonesoup_file2, stonesoup_output));

                stonesoup_threadInput = new StringBuilder().append(stonesoup_taint);

                stonesoup_output.println("Info: Spawning thread 1.");
                stonesoup_thread1.start();

                stonesoup_output.println("Info: Spawning thread 2.");
                stonesoup_thread2.start();

                try {
                    Tracer.tracepointMessage("Joining threads");
                    Tracer.tracepointMessage("Joining thread-01");
                    stonesoup_thread1.join();
                    Tracer.tracepointMessage("Joined thread-01");
                    Tracer.tracepointMessage("Joining thread-02");
                    stonesoup_thread2.join();
                    Tracer.tracepointMessage("Joined thread-02");
                    Tracer.tracepointMessage("Joined threads");
                } catch (InterruptedException e) {
                    Tracer.tracepointError(e.getClass().getName() + ": " +
e.getMessage());
                    stonesoup_output.println("Interrupted");
                }
                stonesoup_output.println("Info: Threads ended");

                Tracer.tracepointWeaknessEnd();
            }
        }
    }
```

```
/**
 * @param args
 */
public static void main(String[] args) {
    if (args.length == 0) {
        throw new IllegalArgumentException(
                "No arguments provided on command-line.");
    }
    String sourceTaint = args[0];
    if (sourceTaint == null) {
        throw new IllegalArgumentException(
                "No source taint provided in environment variable args[0].");
    }
    weakness(sourceTaint, System.err);
}
```

```
}
```

J - CWE-832A Source Code

```
package com.pontetec.weakness.concurrency;

import java.io.PrintStream;
import java.util.concurrent.locks.ReentrantLock;
import com.pontetec.stonesoup.trace.Tracer;

public class CWE832A {
    /*
     * This test takes a string.  It locks a lock upon encountering the first
     * a/A in the string, using one lock for lower case a and a different lock
     * for upper case A.  Upon encountering a second a/A in the string, it unlocks
     * the lower case lock.  If an A appears before an a, then the lock is
     * unlocked without previously being locked, resulting in an exception.
     */
    public static class HelloRunnable implements Runnable {
        private static ReentrantLock upperLock;
        private static ReentrantLock lowerLock;
        private static int count;

        private String input;
        private PrintStream output;

        public int getCount() {
            Tracer.tracepointLocation("STONESOUP_INJECTED_FILE",
"HelloRunable.getCount");
            return count;
        }

        private void lockA(Character cc) {
            Tracer.tracepointLocation("STONESOUP_INJECTED_FILE",
"HelloRunable.lockA");
            Tracer.tracepointMessage("CROSSOVER-POINT: BEFORE");
            /* STONESOUP: CROSS-OVER POINT (Unlock of a Resource that is not Locked)
*/
            if (Character.isUpperCase(cc)) {
                Tracer.tracepointMessage("Locking upperLock");
                upperLock.lock();
            } else {
                Tracer.tracepointMessage("Locking lowerLock");
                lowerLock.lock();
            }
            Tracer.tracepointMessage("CROSSOVER-POINT: AFTER");
        }

        private void unlockA(Character cc) {
            Tracer.tracepointLocation("STONESOUP_INJECTED_FILE",
"HelloRunable.unlockA");
            Tracer.tracepointMessage("TRIGGER-POINT: BEFORE");
            /* STONESOUP: TRIGGER POINT (Unlock of a Resource that is not Locked) */
            Tracer.tracepointMessage("Unlocking lowerlock");
            lowerLock.unlock();
```

```
            Tracer.tracepointMessage("TRIGGER-POINT: AFTER");
        }

        private void cleanLocks() {
            Tracer.tracepointLocation("STONESOUP_INJECTED_FILE",
"HelloRunable.cleanLocks");
            if (upperLock.isHeldByCurrentThread()) {
                Tracer.tracepointMessage("Unlocking upperLock");
                upperLock.unlock();
            }
            if (lowerLock.isHeldByCurrentThread()) {
                Tracer.tracepointMessage("Unlocking lowerLock");
                lowerLock.unlock();
            }
        }


        public void run() {
            Tracer.tracepointLocation("STONESOUP_INJECTED_FILE", "HelloRunable.run");

            try {
                int index = 0;
                while (index < input.length()) {
                    char cc = input.charAt(index);
                    index ++;
                    if (Character.toUpperCase(cc) == 'A') {
                        lockA(cc);
                        break;
                    }

                }

                while (index < input.length()) {
                    char cc = input.charAt(index);
                    index++;
                    if (Character.toUpperCase(cc) == 'A') {
                        unlockA(cc);
                        break;
                    } else {
                        count++;
                    }
                }

                cleanLocks();
                output.println(
                        "Info: Found " + getCount() + " letters between a and a");
            }
            catch (java.lang.RuntimeException e) {
                e.printStackTrace(output);
                throw e;
            }
        }


        public HelloRunnable(String input, PrintStream output) {
            Tracer.tracepointLocation("STONESOUP_INJECTED_FILE", "HelloRunable.ctor");
            upperLock = new ReentrantLock();
            lowerLock = new ReentrantLock();
            count = 0;
```

```
            this.input = input;
            this.output = output;
        }
    }

    public static void weakness(final String stonesoup_value, PrintStream
stonesoup_output) {
        Tracer.tracepointWeaknessStart("CWE832", "A", "Unlock of a Resource that is
not Locked");

        Tracer.tracepointMessage("Creating thread");

        Thread stonesoup_thread1 = new Thread(new HelloRunnable(stonesoup_value,
stonesoup_output));

        stonesoup_thread1.start();
        try {
            Tracer.tracepointMessage("Joining thread-01");
            stonesoup_thread1.join();
            Tracer.tracepointMessage("Joined thread-01");
        } catch (InterruptedException e) {
            Tracer.tracepointError(e.getClass().getName() + ": " + e.getMessage());
            stonesoup_output.println("Interrupted");
        }
        stonesoup_output.println("Info: Thread ended");

        Tracer.tracepointWeaknessEnd();
    }

    /**
     * @param args
     */
    public static void main(String[] args){
        if (args.length == 0) {
            throw new IllegalArgumentException(
                    "No arguments provided on command-line.");
        }
        String sourceTaint = args[0];
        if (sourceTaint == null) {
            throw new IllegalArgumentException(
                "No source taint provided in environment variable args[0].");
        }
        weakness(sourceTaint, System.err);
    }
```

```
}
```

J - CWE-833A Source Code

```
package com.pontetec.weakness.concurrency;

import java.io.PrintStream;
import java.util.concurrent.locks.ReentrantLock;
import com.pontetec.stonesoup.trace.Tracer;

public class CWE833A {
    /*
     * This test takes a string.  If the first letter is a lower case letter,
     * the main thread will get a lock guarding an integer, and count the
     * number of lower-case letters into that integer.    A second thread will
     * get a lock guarding a second integer, and will count the number of
     * upper-case letters into that integer.\n    If the first letter is an upper
     * case letter, both threads will attempt to get the same lock, resulting
     * in a deadlock.
     */
    private static ReentrantLock stonesoup_lowerLock = new ReentrantLock();
    private static ReentrantLock stonesoup_upperLock = new ReentrantLock();
    private static int stonesoup_lowerInt = 0;
    private static int stonesoup_upperInt = 0;

    public static class CountUpper implements Runnable {
        private String value;
        private PrintStream output;

        public void run() {
            Tracer.tracepointLocation("STONESOUP_INJECTED_FILE", "CountUpper.run");

            Tracer.tracepointMessage("TRIGGER-POINT: BEFORE");
            /* STONESOUP: TRIGGER POINT (Deadlock) */
            Tracer.tracepointMessage("Locking lock");
            stonesoup_upperLock.lock();
            Tracer.tracepointMessage("TRIGGER-POINT: AFTER");
            try {
                for (int ii = 0; ii < value.length(); ii++) {
                    if (Character.isUpperCase(value.charAt(ii))) {
                        stonesoup_upperInt += 1;
                    }
                }
            } finally {
                Tracer.tracepointMessage("Unlocking lock");
                stonesoup_upperLock.unlock();
            }
            output.println("Info: Thread ending, upperInt " + stonesoup_upperInt);
        }

        public CountUpper(String value, PrintStream output) {
            Tracer.tracepointLocation("STONESOUP_INJECTED_FILE", "CountUpper.ctor");
            this.value = value;
            this.output = output;
        }
```

```
    }

    private static boolean firstIsUpper(String value) {
        Tracer.tracepointLocation("STONESOUP_INJECTED_FILE", "firstIsUpper");
        return (Character.isUpperCase(value.charAt(0)));
    }


    public static void weakness(final String stonesoup_value, PrintStream
stonesoup_output) {
        Tracer.tracepointWeaknessStart("CWE833", "A", "Deadlock");

        Tracer.tracepointVariableString("stonesoup_value", stonesoup_value);

        boolean stonesoup_upper = firstIsUpper(stonesoup_value);
        ReentrantLock stonesoup_lock = null;

        Tracer.tracepointMessage("CROSSOVER-POINT: BEFORE");
        /* STONESOUP: CROSS-OVER POINT (Deadlock) */
        if (stonesoup_upper) {
            Tracer.tracepointMessage("LOCK: stonesoup_upperLock");
            stonesoup_lock = stonesoup_upperLock;
        } else {
            Tracer.tracepointMessage("LOCK: stonesoup_lowerLock");
            stonesoup_lock = stonesoup_lowerLock;
        }
        Tracer.tracepointMessage("Locking lock");
        stonesoup_lock.lock();
        Tracer.tracepointMessage("CROSSOVER-POINT: AFTER");

        try {
            Tracer.tracepointMessage("Creating thread");

            Thread stonesoup_thread1 = new Thread(new CountUpper(stonesoup_value,
stonesoup_output));

            stonesoup_thread1.start();

            for (int ii = 0; ii < stonesoup_value.length(); ii ++) {
                if (stonesoup_upper &&
Character.isUpperCase(stonesoup_value.charAt(ii))) {
                    stonesoup_upperInt += 1;
                } else if (!stonesoup_upper &&
!Character.isUpperCase(stonesoup_value.charAt(ii))) {
                    stonesoup_lowerInt += 1;
                }
            }

            try {
                Tracer.tracepointMessage("Joining thread-01");
                stonesoup_thread1.join();
                Tracer.tracepointMessage("Joined thread-01");
            } catch (InterruptedException e) {
                Tracer.tracepointError(e.getClass().getName() + ": " +
e.getMessage());
                stonesoup_output.println("Interrupted");
            }
        } finally {
            Tracer.tracepointMessage("Unlocking lock");
```

```
            stonesoup_lock.unlock();
        }
        stonesoup_output.println("finished evaluating");
        stonesoup_output.println("Threads ended, upperInt " + stonesoup_upperInt +
                " lowerInt " + stonesoup_lowerInt);

        Tracer.tracepointWeaknessEnd();
    }

    /**
     * @param args
     */
    public static void main(String[] args){
        if (args.length == 0) {
            throw new IllegalArgumentException(
                    "No arguments provided on command-line.");
        }
        String sourceTaint = args[0];
        if (sourceTaint == null) {
            throw new IllegalArgumentException(
                "No source taint provided in environment variable args[0].");
        }

        weakness(sourceTaint, System.err);
    }
```

```
}
```

J - CWE-839A Source Code

```
package com.pontetec.weakness.numberhandling;

import java.io.PrintStream;
import java.util.ArrayList;
import java.util.List;
import com.pontetec.stonesoup.trace.Tracer;

public class CWE839A {
    public static void weakness(final int value, PrintStream output) {
        /*
         * CWE-839 Numeric Range Comparison Without Minimum Check
         * This test takes in a value from an argument and uses it
         * as a array accessor resulting in an ArrayIndexOutOfBounds
         * exception.
         */
        // Good values: >= 0
        // Bad values:     < 0

        Tracer.tracepointWeaknessStart("CWE839", "A", "Numeric Range Comparison
Without Minimum Check");

        @SuppressWarnings("serial")
        List<String> stonesoup_face_cards = new ArrayList<String>() {{
            add("Hearts (Jack)");
            add("Hearts (Queen)");
            add("Hearts (King)");
            add("Hearts (Ace)");
            add("Clubs (Jack)");
            add("Clubs (Queen)");
            add("Clubs (King)");
            add("Clubs (Ace)");
            add("Spades (Jack)");
            add("Spades (Queen)");
            add("Spades (King)");
            add("Spades (Ace)");
            add("Diamonds (Jack)");
            add("Diamonds (Queen)");
            add("Diamonds (King)");
            add("Diamonds (Ace)");
            add("Joker");
            add("Joker");
        }};

        Tracer.tracepointVariableInt("value", value);
        Tracer.tracepointVariableInt("stonesoup_face_cards.size()",
stonesoup_face_cards.size());
        Tracer.tracepointMessage("CROSSOVER-POINT: BEFORE");
        /* STONESOUP: crossover point (no minimum check) */
        if (value >= stonesoup_face_cards.size()) {
            Tracer.tracepointMessage("CROSSOVER-POINT: AFTER");
            output.printf("Card not available for %d.\n", value);
```

```
        } else {
            Tracer.tracepointMessage("CROSSOVER-POINT: AFTER");
            try {
                Tracer.tracepointMessage("TRIGGER-POINT: BEFORE");
                /* STONESOUP: trigger point (lookup with negative value) */
                output.printf("Selected Card = %s\n",
stonesoup_face_cards.get(value));
                Tracer.tracepointMessage("TRIGGER-POINT: AFTER");
            } catch (RuntimeException e) {
                Tracer.tracepointError(e.getClass().getName() + ": " +
e.getMessage());
                e.printStackTrace(output);
                throw e;
            }
        }
        Tracer.tracepointWeaknessEnd();
    }

    /**
     * @param args
     */
    public static void main(String[] args) {
        int value = 0;
        String sourceTaint = args[0];
        if (sourceTaint == null) {
            throw new IllegalArgumentException(
                    "No source taint provided in environment variable args[0].");
        }
        try {
            value = Integer.parseInt(sourceTaint);
        } catch (NumberFormatException e) {
            throw new IllegalArgumentException(
                    "Source taint is not a valid int.", e);
        }
        weakness(value, System.err);
    }
```

```
}
```

Appendix B: Supporting Scripts

Some weakness require external scripts to support exercising the benign or exploiting execution paths. In the cases where a script is used across multiple weakness variants, the source code has been provided here as a reference.

runFifos.py

The runFifos.py script was developed and tested against Python 2.7.5. While other versions of Python may work, they are not explicitly supported.

```
#!/usr/bin/env python2.7

import os
import sys
import time
import threading
import argparse

from threading import Thread

ready_count = 0

def t1Handler(event1, event2, file1, syncFlag):
    global ready_count
    os.mkfifo(file1)
    fifo = open(file1, "w")
    fifo.write("abcdefghijklmnopqrstuvwxyzabcdef")

    print("t1: wrote first 32 bytes to file")
    if syncFlag:
        event1.clear()
        ready_count += 1
        if ready_count < 2:
            if event1.wait(2) == False:
                print("t1: timeout occured waiting for t2 to start")
            else:
                print("ALL: threads synchronized")

        event1.set()
    else:
        time.sleep(2)
        print("t1: no synchronization, continuing")
    try:
        fifo.write("abcdefghijklmnopqrstuvwxyzabcdef")
        fifo.close()
    except IOError, e:
        fifo.close()
    os.remove(file1)

    print("t1: wrote second 32 bytes to file")
    if syncFlag:
        event2.set()
```

```
def t2Handler(event1, event2, file2, syncFlag):
    global ready_count
    os.mkfifo(file2)
    fifo = open(file2, "w")
    fifo.write("12345678901234567890123456789012")

    print("t2: wrote first 32 bytes to file")
    if syncFlag:
        event1.clear()
        ready_count += 1
        if ready_count < 2:
            if event1.wait(2) == False:
                print("t2: timeout occured waiting for t1 to start")
            else:
                print("ALL: threads synchronized")

        event1.set()
        if event2.wait(2) == False:
            print("t2: timeout occured waiting for t1 to finish")
        else:
            print("t2: t1 finished")
            time.sleep(2)

    else:
        time.sleep(2)
        print("t2: no synchronization, continuing")
    try:
        fifo.write("12345678901234567890123456789012")
        fifo.close()
    except IOError, e:
        fifo.close()
    print("t2: wrote second 32 bytes to file")
    os.remove(file2)

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument("fifo", nargs="+", help="Fifo files")
    parser.add_argument("--nosync", dest='syncFlag', action='store_false',
required=False, help="Flag for thread synchronization")
    parser.set_defaults(syncFlag=True)
    args = parser.parse_args()
    event1 = threading.Event()
    event2 = threading.Event()
    t1 = Thread(target=t1Handler, args=(event1, event2, args.fifo[0], args.syncFlag))
    if len(args.fifo) > 1:
        t2 = Thread(target=t2Handler, args=(event1, event2, args.fifo[1],
args.syncFlag))

    t1.start()
    if len(args.fifo) > 1:
        t2.start()
    t1.join()
    if len(args.fifo) > 1:
        t2.join()

if __name__ == "__main__":
```

```
main()
```