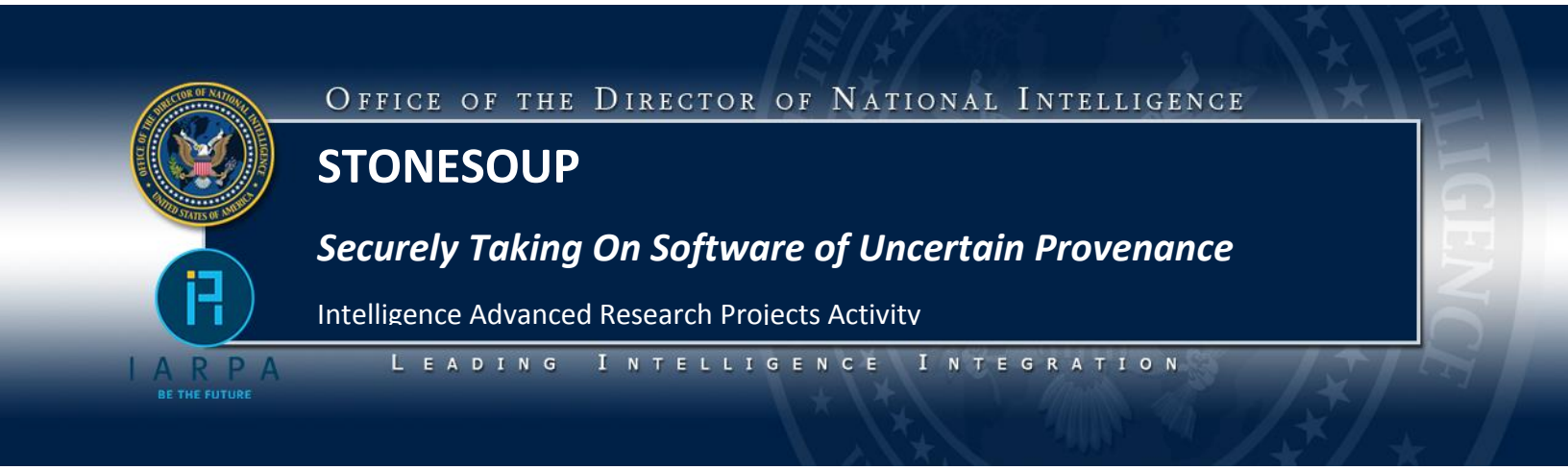

STONESOUP Phase 3

Test Generation Report

12 December 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.

Table of Contents

List of Figures

List of Tables

1 Executive Summary

The Intelligence Advanced Research Projects Activity (IARPA) project STONESOUP (Securely Taking On New Executable Software Of Uncertain Provenance) aimed to eliminate the effects of vulnerabilities in software applications by (a) extending the scope and capability of approaches for analysis, confinement, and diversification; (b) addressing a wide range of security vulnerabilities within the same framework; and (c) integrating approaches to leverage the strengths and weaknesses of each. The program aimed to provide comprehensive, automated techniques for vulnerability reduction in software of uncertain provenance.

To determine the effectiveness of the performer technology at mitigating software vulnerabilities, the STONESOUP Test and Evaluation Team (T&E) developed an automated system to create, run, and evaluate thousands of test cases using performer technology. In this effort, the Test Generation Team developed thousands of programs each with a known vulnerability that could be exploited by a user. The T&E Team developed the Test and Evaluation eXecution and Analysis System (TEXAS) that could run these thousands of programs with (or without) performer technology and evaluate their effectiveness at mitigating the vulnerabilities.

This report discusses how T&E implemented the tasks described in the STONESOUP Phase 3 Test Data Generation Plan (TGP). The TGP describes the composition of test cases, and includes the test case naming standard, which allows a user to identify the behavior of each test case based on the name of the test case.

The TGP outlines a plan for testing the tools developed by each of the three performer teams that were selected to enter into Phase 3 of STONESOUP. The prime contractors for the three teams, and the operating system and language they each addressed are listed in Table 1.

Table 1 Performer teams

Performer	OS	Language
Columbia	CentOS 6.5	C
Grammatech	Ubuntu 12.04	Binary from C source code
Kestrel	Ubuntu 12.04	Java

The Test Generation Team developed systems to automatically inject vulnerabilities into particular base programs. These vulnerabilities consisted of a taint source, 3 code complexity features, and a weakness that corresponded to one of the Common Weakness Enumerations (CWEs) developed by MITRE. The Test Generation Team further developed

a packager to create instructions for building, running, and evaluating test cases in an automated fashion.

T&E developed the taints sources as described in Section 3.4.1 of the TGP. The TGP indicates that socket taint source will only be used with server programs. This requirement was found not to be necessary, and so T&E used socket taint source with all base programs.

T&E developed the control flow, data flow, and data type complexity features as described in Section 3.4 of the TGP.

T&E developed the weaknesses as described in Section 3.2 of the TGP. However, some of the performer teams, with the agreement of the customer, changed which weakness classes their tools addressed. Accordingly, T&E altered which weaknesses were developed to address these changes. In addition, some weaknesses specified in the TGP were not developed because they were fully covered by other weaknesses, did not apply to the target language, or could not be tested in an automated fashion. Details on these changes are provided in Section 10.

T&E developed one or more weakness snippets that each exercised a flaw identified by a particular Common Weakness Enumeration (CWE). Table 2 shows the number of Common Weakness Enumerations (CWEs) for which snippets were developed for each weakness class for each language. All weakness snippets developed by T&E were validated by the Independent Verification & Validation Team. T&E addressed all errors and concerns raised by IV&V regarding weakness snippets.

Table 2 Weakness Classes Mapped to Language and Common Weakness Enumerations

Weakness Class	# CWEs	
	C/Binary	Java
Number Handling	9	8
Tainted Data	N/A	6
Error Handling	N/A	8
Resource Drain	11	9
Injection	3	4
Concurrency Handling	15	15
Memory Corruption	17	N/A
Null Pointer Error	1	N/A

T&E injected faults into base programs in both C and Java. The programs chosen for Phase 3 are specified in Section 5.4. At the request of the customer, T&E chose base programs for Phase 3 to achieve an average of 500,000 lines of code. The C base programs had an

average of 444,429 lines of code, and the Java base programs had an average of 932,825 lines of code.

In addition, T&E injected vulnerabilities into 4 other programs to aid in testing the injection framework, taint sources, and weaknesses. These additional programs are the small programs C-Tree and J-Tree, and the Phase 2 programs Grep and JMeter. These programs are discussed in Section 5.4.

In accordance with Section 4 of the TGP, T&E developed 531 test cases for each weakness case in C, 637 for each weakness class in Binary (which are implemented in C), and 478 test cases for each weakness class in Java. The test case specifications are drawn uniformly from the population of possible test cases, as described in Section 0 of this report and in the STONESOUP Phase 3 Test and Evaluation Final Report.

T&E ran each test case under two conditions – one with no performer technology present (e.g., Stage One) and one with the performer technology under evaluation present (e.g., Stage Two). IV&V validated which test cases ran successfully in Stage One.

2 Overview of Test Generation

T&E obviously could not hand-create thousands of different large (500,000 lines of code) programs each with a single known weakness. So T&E developed a process that relied on existing large code bases to automatically generate test cases. This involved multiple areas of effort:

1. Selecting and modifying base programs
2. Developing weakness code with benign and exploiting inputs
3. Injecting weakness code into the base program
4. Packaging test cases

The following sections provide an overview of these areas.

2.1 Select and Modify Base Programs

One of the goals of STONESOUP was to develop mitigation strategies for large-scale, real-world programs. It was not sufficient for T&E to create toy programs with vulnerabilities—the performer technology had to be tested against large, complicated programs. Accordingly, T&E searched open source software corpuses to find programs that could be used as a base into which to inject vulnerabilities. T&E performed the following steps to select and modify base programs:

1. Identify possible base programs
2. Create scripts to build the base programs from source
3. Modify the base programs so they could be built with the injection frameworks
4. Modify the base programs to respect library environment variables
5. Verify identity translation for the base programs
6. Inject function logging statements into each base program, producing log-injected base programs
7. For each base program, build a skeleton directory structure, create an XML file containing build and run instructions, and identify 10 different inputs to base program
8. Run log-injected base programs on each of the 10 inputs
9. Determine functions used by all inputs for each base program
10. Select 10 injection points from the common set for each base program

2.2 Develop Weakness Code

T&E used the Common Weakness Enumerations (CWE) defined by MITRE as a guide in creating 98 C weaknesses and 60 Java weaknesses that each cause unintended program behavior. These weaknesses existed as snippets of code that could be inserted into the Abstract Syntax Tree (AST) of the base programs.

For each weakness, T&E also developed inputs that resulted in benign vulnerability behavior and inputs that resulted in exploit vulnerability behavior. Exploit inputs would cause negative technical impacts such as program crash, thread deadlock, or private information exposure. The benign and exploiting inputs were stored in a YAML file that could be used by automated systems to create test cases. In addition, the YAML file stored information about other processes that were required to run the weakness. These processes were tightly coupled to the weakness snippet; they might set particular environment variables or run particular scripts at particular times to trigger the appropriate weakness behavior.

For each weakness, T&E also developed an automated test to determine whether the exploiting input triggered the expected negative technical impact. Just as it was not possible for T&E to hand-create thousands of large programs with a single weakness, it was not possible for T&E to hand-verify the results of thousands of test cases. To address this problem, T&E developed "observables" for the exploit condition of each weakness that could be used by an automated test system to determine if the exploit had occurred.

2.3 Inject Weakness Code

T&E developed software to inject weaknesses into base programs. For C and Binary programs, this software relied on the ROSE Compiler Infrastructure for Abstract Syntax Tree (AST) manipulation. For Java programs, a separate system was developed that relied on the Eclipse Java Development Toolkit (JDT) for AST manipulation.

T&E injected several separate pieces of code into the base programs to accommodate the needs of automated testing while imitating the complexity of vulnerabilities found in the wild. In particular, T&E developed and injected into each base program:

1. An atomic barrier to ensure that the vulnerability was run no more than once
2. A taint source that allowed the user to insert data into the program
3. Three code complexity features—one each of control flow, data flow, and data type—to obfuscate the vulnerability
4. A weakness that acts in either a benign or exploit fashion depending upon user input

Figure 1 shows the code injected into each base program.

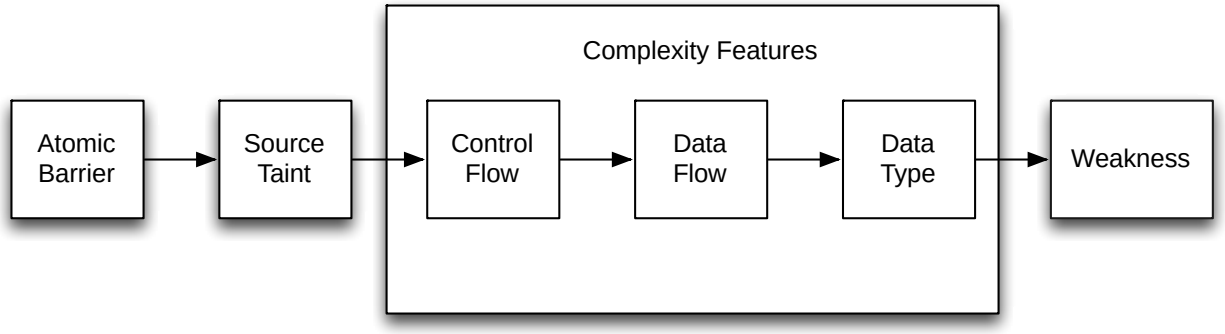

Figure 1 Code injected into each base program

T&E performed extensive unit testing to ensure that these different components could work together seamlessly in the automated test system. In particular, T&E tested all possible combinations of taint sources and code complexity features, resolved bugs where possible, and made adjustments for unavoidable incompatibilities.

2.4 Package Test Cases

T&E developed a packaging system (found in the ss_testcases repository) that took as input a test case name and produced a compressed tar file containing a test case. Each test case contained a base program with an injected weakness, a skeleton directory structure to use in running the base program with the weakness, and XML that instructed TEXAS how to run the test case. The packager also added output checks for exploit inputs and added information about the technical impact of each weakness.

Figure 2 shows the inputs to the packager and the components of the packaged test case.

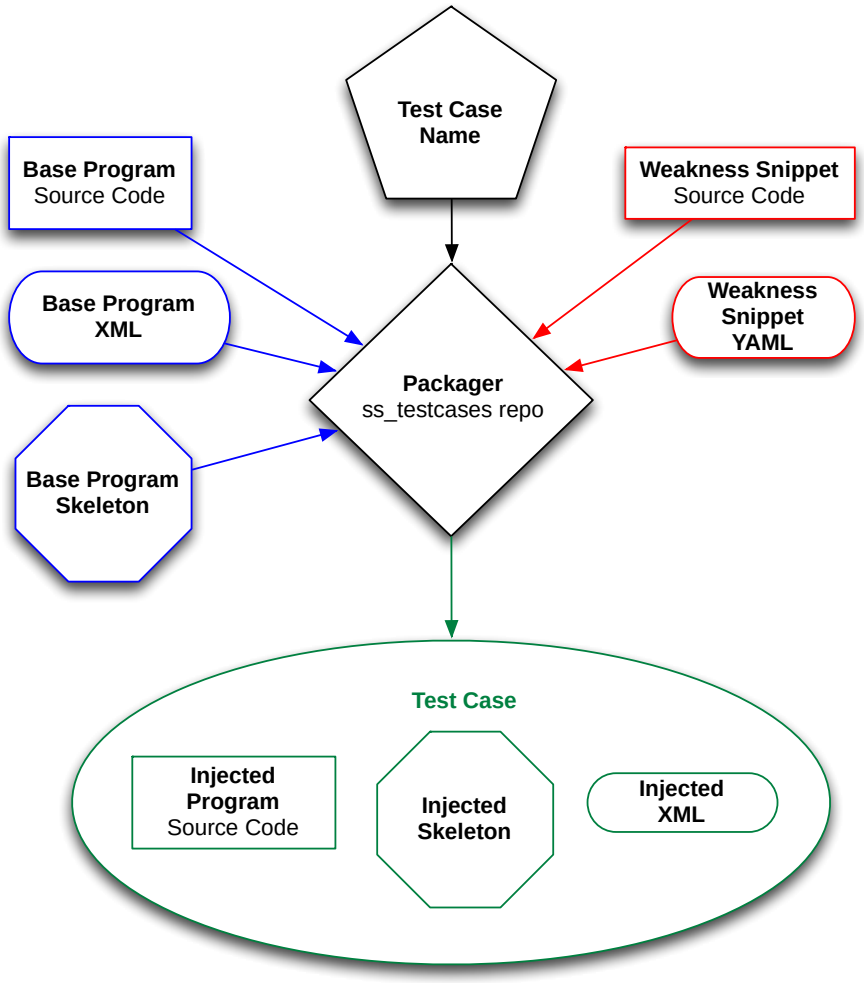

Figure 2 Test Case Packager

3 Selecting and Modifying Base Programs

To develop test cases, T&E began by identifying candidate base programs into which to insert weaknesses. These base programs had to have the following characteristics:

- Open source so T&E could have access to source code
- Written in the appropriate language—C or Java
- Function as a stand-alone program—not solely a library for use by other software
- Have the appropriate number of lines of code—for Phase 3, the goal was an average of 500,000 lines of code
- For C/Binary programs, buildable with gcc (not clang), for compatibility with the ROSE compiler used to manipulate C abstract syntax trees
- For Java programs, buildable using the Ant build system (not Maven) for compatibility with the injection system

To ensure a variety of test situations, the group of programs chosen included:

- Client programs and server programs
- Command-line driven programs and GUI-driven programs.

3.1 Candidate base programs

Over the course of the project, T&E evaluated a large set of candidate programs as possible base programs. For C/Binary, these included:

- Busybox – tool combining common UNIX utilities into a single small executable
- Cherokee – web server
- Claws Mail – email and news client
- D-Bus – inter-process communication system
- Diff – data comparison utility
- Exim – mail transter agent
- FFmpeg – multi-media data processor
- FTP Server – file transfer server
- GIMP – image manipulation software
- Grep – file search tool
- ImageMagick – bitmap manipulation tool
- Irssi – IRC client
- Mutt – email client
- Nginx – reverse proxy server for HTTP
- OpenSSL – cryptography
- Pidgin – chat client
- Postfix – mail transfer agent
- PostgreSQL – relational database

- SDCC – small device C compiler
- Subversion – version control system
- Sudo – tool for running program as a different user
- TCPDump – command-line packet analyzer
- Vim – editor
- Wget – command-line tool for retrieving files from web
- Wireshark – network protocol analyzer
- WWW –W3C browser
- Zsh – shell

For Java, these included:

- Ant – Apache Java build tool
- Barcode4J – generator for barcodes
- CoffeeMud - MUD game engine
- Derby – Apache relational database
- Elasticsearch – search and analytics engine
- FindBugs – static code analysis tool
- Google Web Toolkit – web development tool
- Hadoop – Apache big data storage and processing framework
- HTML Cleaner – HTML to XML transformation
- James – Apache mail server
- Jena – semantic web framework
- JMeter – Apache load testing application
- Jtest – automated Java software testing and static analysis
- Lenya – Apache XML content management system
- Lucene – Apache search software
- Maven – Apache Java build tool
- OpenDS – directory service
- PMD – source code analyzer
- POI – Apache tool for accessing Microsoft Office documents
- SchemaSpy – generates graphical representations of tables
- Tomcat – Apache web server

3.2 Scripting the build process

After generating a candidate set of programs, T&E then developed an automated process to build each program on the appropriate operating systems – CentOS 6.5 and Ubuntu 12.04 for C/Binary programs, and Ubuntu 12.04 for Java programs. For C/Binary, these build scripts required using gcc and the appropriate build system for the package (e.g., make, cmake, imake). For Java, this required using Javac and the Ant build system.

For each base program, T&E identified other software on which the program depended, downloaded the source for those dependencies, and developed scripts to build the dependencies. This process continued until T&E reached a core set of programs from the standard distributions, such as bash and the X windows system, that were assumed to be already installed.

After identifying dependencies and creating build scripts for the dependencies, T&E developed build scripts for the candidate base programs themselves, using the same basic build process that would be used by the base program developers.

4 Developing the Injection Framework

The STONESOUP Phase 3 Test Generation Plan required T&E to inject weakness code into both C and Java programs. T&E thus had to develop injection frameworks for both languages. These injection frameworks read in source code and converted it into an Abstract Syntax Tree (AST) in memory, performed some translation on it by adding nodes to the AST, and then output modified source code.

4.1 Code Injection Overview

The following definitions are useful in understanding the injection frameworks:

- Abstract syntax tree: An internal representation of a programming language where each node of the tree corresponds to a feature appearing in the source code. Abstract syntax trees are generated by any program that acts on source code, including compilers translating from source code to object code, integrated development environments allowing the programmer to manipulate code, and tools allowing manipulation of source code.
- AST generator: A tool that creates and manipulates an abstract syntax tree, either by translating source code into an abstract syntax tree or by creating an abstract syntax tree programmatically without pre-existing source code. T&E used the ROSE Compiler Infrastructure as the AST generator for C. It used the Eclipse Java Development Toolkit (JDT) as the AST generator for Java.
- AST unparser: A tool that translates an abstract syntax tree back into source code. In practice, this was simply an API call to the AST generator that outputs the (modified) abstract syntax tree in the original source language.
- Original software: The source code that would be modified by this system
- Payload software: The code that would be inserted into the original source code by the system. The payload software may:
    a) Exist as source code that was then translated into an abstract syntax tree, or
    b) Exist as an abstract syntax tree that was generated programmatically, or
    c) Be a combination of both types above.
- Modified software: The original software with the payload software inserted into it. This modified software could exist either as an abstract syntax tree, or as source code containing both the original source code and the inserted payload(s).
- Injection point: The location within the original abstract syntax tree where a payload abstract syntax tree would be inserted

Figure 3 shows a simple use of the injection framework, where a single payload was injected at a single point into a single source file. In this figure, the original source code was run through an AST generator to create an abstract syntax tree. A payload abstract syntax tree was created either by running payload source code through an AST generator or by

creating an abstract syntax tree for the payload programmatically. The original abstract syntax tree was combined with the payload abstract syntax tree to create a modified abstract syntax tree. This modified abstract syntax tree was sent to an AST unparser to output modified source code that contains both the original source code and the payload.

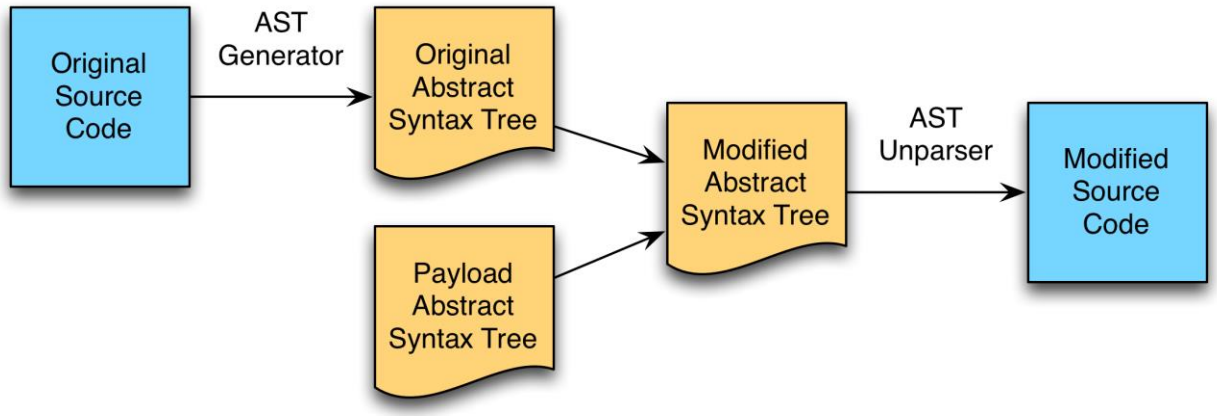

Figure 3 Injection Framework Flow Chart – Single Source, Single Payload

Figure 4 shows a more complicated case, where the injection framework generated an AST from multiple original source files, injected multiple payloads into multiple injection points within that AST, and then output multiple modified files.

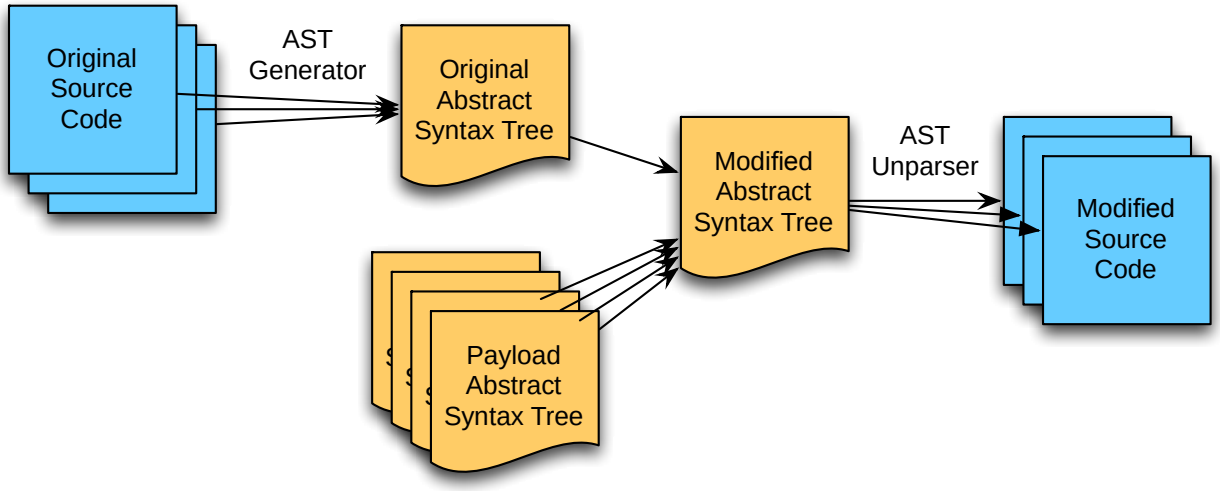

Figure 4 Injection Framework Flow Chart - Multiple Sources, Multiple Payloads

4.2 Specific Injection Frameworks

For C and Binary programs, T&E developed a tool called ss_vuln_injector that used the ROSE Compiler Infrastructure from Lawrence Livermore National Laboratory to create and manipulate abstract syntax trees. For Java, T&E developed a tool called ss_vinject4j that

used the Eclipse Java Development Tools (JDT) API. ss_vuln_injector and ss_vinject4j read in a C or Java program and converted it into an AST using the appropriate backend. They optionally altered the code using the AST system, and then wrote out source code reflecting any changes made to the AST.

The injection frameworks ss_vuln_injector and ss_vinject4j performed the following steps to complete an injection:

- Read in the base program, converting it to an AST
- Located the injection point in the AST
- Inserted into the AST an atomic barrier to ensure the vulnerability was run only one time
- Inserted into the AST a taint source, allowing the user to provide input to the vulnerability
- Inserted one or more code complexity features to obfuscate the data and control flows
- Inserted a weakness that can have either benign or exploit behavior, depending upon the user input
- Read in the YAML file specifying benign and exploit inputs for the weakness
- Output the injected AST as source code
- Output the injected YAML file, containing details about how to create inputs for this particular combination of taint source and weakness

Figure 5 indicates the inputs and outputs of the injection systems for C and Java.

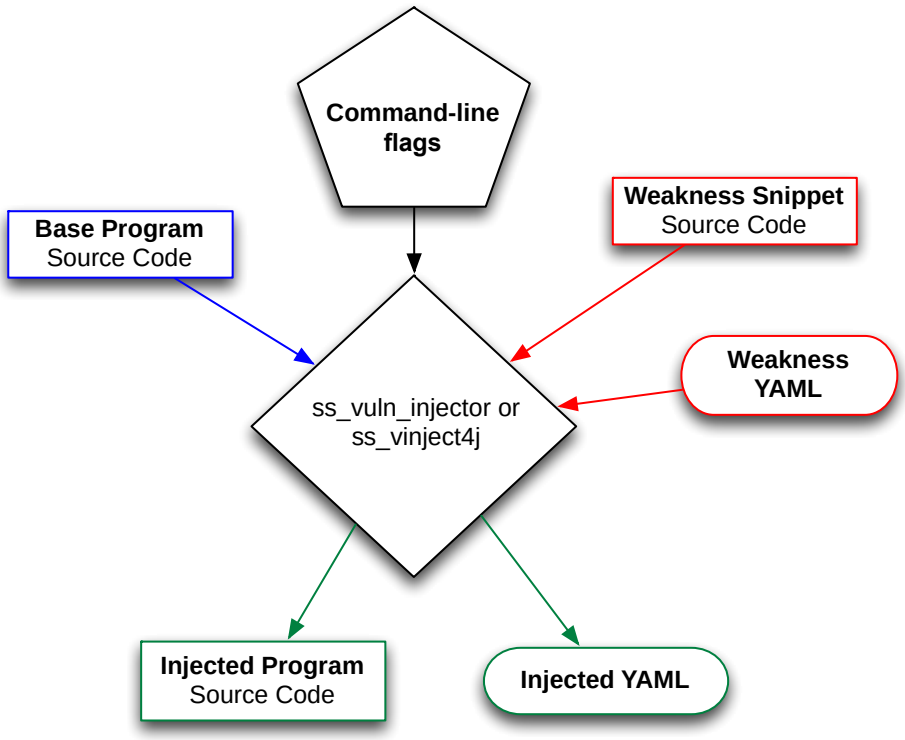

Figure 5 Inputs and outputs for injection system

4.3 Atomic Barrier

For accurate test results, vulnerabilities injected into a base program must run only one time. This presented a problem, because code might be injected into frequently-used utility functions such as IP address format functions. These functions could potentially be called millions of times per second, creating obvious problems for the performance of the injected code, in addition to uncertainties about exploit behavior due to differing numbers of executions.

To resolve these problems, T&E used atomic barriers to ensure the vulnerability code was run no more than once. For C programs, T&E injected a call to the gcc built-in __sync_bool_compare_and_swap. This built-in atomically sets a variable to a given updated value if the current value equals the expected value, and returns true if the operation was successful.

For C programs, T&E injected a further barrier using the file system. Some of the C server programs (PostgreSQL and Subversion's svnserve) used forking to create multiple processes. If the fork occurred before the atomic synchronization operation, each forked process attempted to independently run the vulnerability. To address this problem, inside the atomic built-in barrier, T&E injected a call to mkdir. The mkdir call attempted to atomically create a directory with a given name. It returned 0 if the call was successful and

1 otherwise. If the mkdir call was successful, the weakness would run, but otherwise it would be skipped. The mkdir operation was attempted only if the atomic built-in succeeded to reduce the number of times the code accesses the file system. The mkdir command was atomic so long as it occurred on a local file system, which was the case for the system T&E used. If it is later desired to use a non-local file system, alternative approaches should be considered.

For Java programs, T&E injected a call to java.util.concurrent.atomic.AtomicReference.compareAndSet. This operation behaves identically to the gcc built-in __sync_bool_compare_and_swap. It atomically sets a variable to a given updated value if the current value equals the expected value, and returns true if the operation was successful.

It was not necessary to use the file system barrier for the Java programs, since the Java Virtual Machine (JVM) itself does not fork new processes unless directed to do so by an exec command. None of the selected Java base programs created child JVMs running the same code, so the atomic barrier within the single JVM was sufficient to ensure single execution. It would be straightforward to add a file system check to the Java injection process if a new base program required it.

4.4 Other Injected Code

All other injected code is discussed in detail in later sections:

- Taint sources are discussed in Section 8
- Code Complexity Features are discussed in Section 9
- Weaknesses are discussed in Section 10

5 Verifying the Identity Translation

After selecting candidate base programs, T&E attempted to perform an identity translation on each candidate program using the injection system. This required modifications to the build process for each candidate base program. For some programs, the build process was sufficiently incompatible with the injection framework that the programs could not be modified to allow an identity translation. These programs were dropped from consideration as Phase 3 base programs.

5.1 Modifying build process to use injection framework

T&E modified the build process for each candidate base program to replace the compiler used by the base program (gcc or javac) with the appropriate injection framework (ss_vuln_injector using ROSE for C, and ss_vinject4j using the Eclipse JDT for Java). In some programs, this was an easy task, involving simply changing an environment variable. However, in most programs, this required hand altering the build process to replace all references to the compiler with calls to the appropriate injection system.

In C, the replacement of the compiler with the injection framework was complicated by the fact that two different compilers are widely used: gcc and clang. The ROSE Injection Framework was built around gcc, and T&E found that it was not possible to replace calls to the clang compiler with calls to the C injection system. As a result of this incompatibility, T&E did not consider programs that were built with clang as candidates for base program injection.

In Java, the replace of javac with the injection framework was complicated by the fact that there are two different build systems: ant and maven. The Java injection tool was written for ant build systems, so a base program that used maven could not be run with it. T&E attempted to use automated tools to convert programs using maven to use ant. However, the automated tools were error-prone and required significant hand-alteration after the fact. Given the size of the base programs and the complexity of their build processes, T&E found it was not possible to convert maven build processes to ant.

5.2 Modifying build process to use external libraries

T&E further modified the build process for each candidate base program to allow arbitrary libraries to be linked to the base program. This modification was necessary because T&E needed to inject into the base programs code that required various external libraries. For example, all weaknesses in both C and Java required access to the Linux Trace Toolkit Next Generation (LTTng) libraries for outputting trace information. Moreover, individual test cases sometimes required other specific libraries. For instance, if a C test case used the socket taint source, the base program would need to link to libmongoose to access the socket functionality. If a C test case used PostgreSQL for a SQL injection weakness, the base

program would need to link to the PostgreSQL library, libpq. If a Java test case used the socket taint source, the base program would need access to the NanoHTTPd Java archive. If a Java test case used PostgreSQL, the base program would need to link to the JDBC4 PostgreSQL Java archive.

Since different test cases needed different libraries, T&E altered the build process for each candidate base program to allow new libraries to be linked as needed. For some base programs, this change was straightforward, as their build process used an environment variable such as LIBS throughout the build process. In these cases, T&E could simply add the new libraries to the appropriate environment variable. However, many base programs did not use standardized environment variables, or used them only for part of their build process. For the majority of base programs, T&E had to debug and alter the build processes to respect LIBS and related variables throughout the code base.

5.3 Performing the identity translation

After altering the build process to use the injection framework and to allow new libraries to be linked in, T&E performed an identity translation on each candidate base program using the injection framework. For this identity translation, no modifications were made to the AST. The code was read in by ss_vuln_injector or ss_vinject4j, converted into an AST, and then output again as source code. Ideally, the code that was output by the injection system should be identical to the code that was input to it, with the exception of whitespace differences. After the identity translation, T&E ran the program again, expecting to see identical behavior to the original code.

In practice, the AST systems do not always accurately perform the identity translation, resulting in various issues that must be addressed.

5.3.1 Translation issues in C

For C/Binary programs, multiple problems arose because of difficulties handling pre-processor directives. In a normal C build process, gcc first pre-processes the code, then compiles it, and then links it. When using the ROSE Compiler Infrastructure to parse the source code, ROSE first pre-processes the code, and then uses the compiler to convert the pre-processed code into an AST. However, for clarity reasons, ROSE does not want to output the pre-processed code. Such code could be hundreds of times as long as the original code and far less readable. ROSE goes to some lengths to try to restore the original code as it appeared before pre-processing. However, if the pre-processor statements were sufficiently complex, ROSE cannot successfully resolve them.

For the C programs, T&E attempted to hand-alter pre-processor directives to accommodate ROSE limitations where possible. However, given the number and complexity of the pre-processor directives, T&E added to the ss_vuln_injector system the capability to skip

transformation of certain problematic files. If the injection system could not successfully transform a file, no point within that file could be used as an injection point for vulnerabilities. This reduced the number of possible injection points, so T&E made every effort to limit the number of skipped files.

5.3.2 Translation issues in Java

Because Java does not use a preprocessor, there were fewer translation issues with the Java injection tool than with the C injection tool. The main translation issues arose from the fact that the Eclipse JDT used a different sequence of operations than the normal Java compiler for type promotion involving the ternary operator. For example, when using the java compiler javac, the following code was valid:

String s = String(test_value ? 0 : “not_valid”);

However, this same code was not valid while using the Eclipse JDT API. Instead, the promotion from int to String must occur before the ternary operator evaluation. For the Eclipse JDT API, the following was required:

String s = test_value ? Integer.toString(0) : “not_valid” ;

T&E had to identify and fix this problem and other similar problems within the Java base programs in order to successfully process the base programs with the ss_vinject4j injection system.

5.4 Phase 3 base programs

The base programs selected for Phase 3 are listed in the Test Data Generation Plan, in Section B-4. We repeat the list here to add information about where to obtain the source code. Note that the base programs typically have to be modified to work with the injection framework. The modified code is available in the ss_base_programs repository.

These programs were selected because they had sufficient lines of code, represented a variety of program types, and identity translation with the injection framework could be made to work for them.

Methods for counting lines of code are inherently controversial—a perfect method does not exist. For STONESOUP, the number of lines of code was determined using the CLOC tool (http://cloc.sourceforge.net/, v1.60). This tool counts actual lines of code, not including blank lines or comment-only lines. For Phase 3, T&E aimed to have base programs with an average of 500,000 lines of code. For the programs actually chosen for Phase 3, the C base programs had an average of 444,429 lines of code, and the Java base programs had an average of 932,825 lines of code.

Table 3 C/Binary Base Programs

Identifier	Base Program	Category	Version	Repository	LOC
ELAS	Elastic Search	Service	1.0.0	http://www.elasticsearch.org/	297,491
CMUD	Coffee MUD	Service	5.8	http://www.coffeemud.org/	537,199
LENY	Apache Lenya	Service/GUI	2.0.4	http://lenya.apache.org/	358,003
LUCE	Apache Lucene	Console	4.5.0	http://lucene.apache.org/	440,299
JENA	Jena	Console	2.11.0	https://jena.apache.org/	377,160
GWTX	Google Web Tookit	GUI	2.6.0-rc3	http://www.gwtproject.org/	656,421

Table 4 Java Base Programs

Identifier	Base Program	Category	Version	Repository	LOC
FFMP	FFMpeg	Console	1.2.2	https://www.ffmpeg.org/	566,908
GIMP	Gimp	GUI	2.8.8	http://www.gimp.org/	711,339
OSSL	OpenSSL	Console	1.0.1e	https://www.openssl.org/	274,204
PSQL	PostgreSQL	Service	9.2.4	http://www.postgresql.org/	731,469
SUBV	Subversion	Console/Service	1.8.3	https://subversion.apache.org/	798,636
WIRE	Wireshark	GUI	1.10.2	https://www.wireshark.org/	2,523,396

In addition, T&E used four small base programs, 2 each in C and Java, for testing the injection system, the taint sources, and the weakness variants. These programs were much smaller than the Phase 3 base programs and provided a simpler system for debugging purposes. They are listed in Table 5.

Table 5 Test Programs

Identifier	Base Program	Category	Version	Repository	LOC
CTREE	C-Tree	Console	1.7.0	http://mama.indstate.edu/users/ice/tree/	2,751
JTREE	J-Tree	Console	-	Written by T&E	284
GREP	Grep	Console	2.14	http://www.gnu.org/software/grep/	47,741
JMET	JMeter	Console	2.8	http://jmeter.apache.org/	103,105

C-Tree (Linux tree utility) and J-Tree (created by the T&E Team) are considered micro-programs because of their small size. Grep and JMeter were selected from the base programs used in Phase 2.

6 Choosing Inputs

After selecting appropriate base programs, T&E developed 10 inputs for each program. It was desired to have the 10 inputs exercise code paths that were as distinct as possible, so that the performer code would have to handle many different cases. To develop these inputs, T&E examined user documentation for the programs, and identified inputs that would exercise as wide a range as possible of user functionality.

As an example, for the relational database PostgreSQL, the 10 inputs performed the following tasks:

1. Initialize a Postgres database
2. Select rows from a table in an existing database
3. Insert a row into a table
4. Create a table
5. Delete a row from a table
6. Drop a table
7. Run a psql script on a database
8. Alter a table by changing the type of a column in a table
9. Select specific columns from a table
10. Select rows from a table using regular expressions

These inputs were specified in the XML file for the PostgreSQL base program.

For details on the inputs used for the other base programs, examine the XML files for those base programs within the ss_base_programs repository. Inputs are often quite complicated, and may require additional data files found in the skeleton directory for the program, especially in the testData and scripts directories.

For each base program input, T&E developed an automated test to detect whether that input had executed successfully. These automated tests were called output checks and were also included in the XML file for the base program.

For 3 of the test programs (C-Tree, J-Tree, and JMeter), T&E developed only 5 inputs. For the remaining test program (Grep), T&E developed 10 inputs. Since these programs were used only for testing the system, and not for inclusion in the test corpus, T&E did not require them to have the full 10 inputs that a base program had.

6.1 Examples of inputs

In the simplest case, an input to a program consisted of command line arguments used when running that program. However, inputs could be far more complicated. If the base program was a server, for example, the server would be started with appropriate arguments, then a separate co-process would be started to make requests to that server. If

the program used a Graphical User Interface (GUI), T&E developed GUI automation scripts to interact with the program.

6.1.1 GUI automation scripts

T&E had to develop a GUI automation solution that:

- Worked on Linux (Ubuntu 12.04, and CentOS 6.5)
- Emulated mouse and keyboard without directly using back-end libraries—this helped integration with performer technology
- Had the ability to tell if a button or screen was available to click on, rather than blindly relying on timing
- Used scripts that could be provided to performer teams

T&E considered the following solutions:

- Xmacro (http://xmacro.sourceforge.net/)
  - Pros: Allowed an X session to be recorded and replayed
  - Cons:
    - Old program, last updated in 2000
    - Written for Ubuntu not CentOS
- AutoKey (https://code.google.com/p/autokey/)
  - Pros: Scripts written in Python
  - Cons:
    - Poor examples when doing complex tasks such as mouse control
    - Crashed on system with no error message, difficult to debug scripts
    - No updates in 2 years
- Sikuli (http://www.sikulix.com/)
  - Pros:
    - Easy scripting, possible to determine if a button existed before attempting to click
    - Used OCR and jython
    - Used java.awt.Robot, which created native mouse and keyboard inputs
  - Cons: Used Java

T&E settled on Sikuli v. 1.1 as the best choice for GUI automation. The programmer would write python scripts that used the Sikuli library to interact with the graphical user interface. The Sikuli library provided functions to check the contents of the screen, identify if a button or window existed before attempting to click on it, provide native mouse clicks and keyboard inputs, and capture screen images. Since the Sikuli scripts were embedded in Python, the programmer had access to the full suite of Python features, and could create relatively robust scripts.

GUI automation was a difficult task, especially when the automation needed to provide repeatable results over tens of thousands of runs. For instance, T&E discovered problems with Ubuntu placing notification windows on the screen as part of standard update management. These notification windows pulled focus to themselves, causing Sikuli mouse clicks and keyboard inputs to be directed away from the intended application. Furthermore, the notification windows altered the appearance of the screen, causing failures in screen content checks. T&E was able to resolve problems as they arose, but the unexpected behavior of GUIs made it difficult to identify all possible problems.

6.2 Examples of output checks

In the simplest case, an output check involved checking the return code of the base program. More complicated checks looked for a particular string (or regular expression) in standard out or standard error when the program was run. Other output checks looked for particular files with particular contents to be created in known locations, or for the base program to complete execution within the timeout window (or not to complete within the timeout window).

The output checks for base programs were specified in the XML file for the base program. These output checks could be arbitrarily nested Boolean expressions, combining the results of multiple different simple output checks.

7 Identifying Injection Points

T&E needed to identify points within the base programs that would execute every time the base program was run on any of the 10 chosen base program inputs. To identify such points, T&E used the injection systems (ss_vuln_injector for C/Binary and ss_vinject4j for Java) to alter the base programs to log every time a function or method was entered. Because of the limitations of the injection frameworks described in Section 1, it was not possible to inject function logging into every file. However, the vast majority of files were injected with these logging methods. After a base program had been injected with a logging method in all (or almost all) functions, the base program was called "log-injected".

T&E then ran each log-injected base program on all 10 base program inputs to generate a list of every function called by each input. Figure 6 provides a simplified graphical depiction (using only 3 base program inputs) of how possible injection points were identified. For each input, T&E generated a log file of all functions called by the base program on that input. These log files correspond to the circles in Figure 6. T&E then took the intersection of these lists of function to obtain the central part of the diagram: the functions executed by every input to the base program. There were typically 10s to 1000s of functions called by every input to the base program.

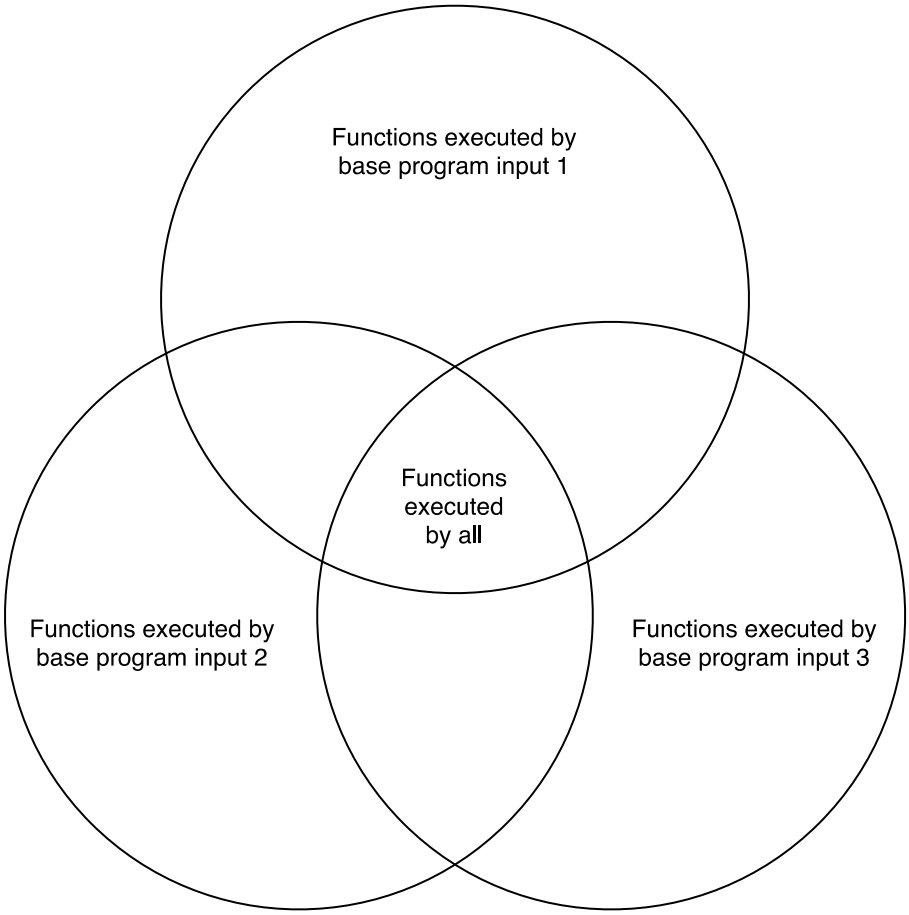

Figure 6 Identifying Possible Injection Points

From this intersected set of functions, 10 functions were chosen to be injection points. When executing the base program on each of the 10 base program inputs, each injection point was called at least one time.

For servers, the injection point could be called before the server loop was established or after. There was no guarantee that the server loop was already running when the injection point was called. It would be possible to add such a guarantee, but it would require additional instrumentation. In particular, code would have to be added to the server loop to set a global flag when the loop has been established and unset the flag when the server loop was exited. The logging functions would have to be changed to only log entries into functions when the global server-loop flag was set. Later, when the base programs run with injected weakness code, the weakness should not fire unless the server-loop flag was set.

After selecting 10 injection points, T&E ran automated tests to further verify that the injection points were actually being executed at run-time. T&E created dummy weaknesses in both C (C-C101B) and Java (J-C101B) that output a special value to standard error when the dummy weakness was run. T&E then altered the packager to have special

behavior when injecting these particular weaknesses. In this case only, the packager would remove any output checks normally used for the base programs, and in their place put a single output check for the special value output by C-C101B and J-C101B. T&E then ran all 10 injection points for all 12 base programs for all 10 input values, and checked that the single specialized output check was satisfied. T&E was thus able to verify that each injection point ran successfully every time it should.

For the test programs (C-Tree, J-Tree, Grep, and JMeter), only one injection point was used, to simplify testing. Since the test programs had a small number of inputs and only a single injection point, T&E verified by hand that the injection point was executed for each of the inputs.

8 Developing Taint Sources

In accordance with the Test Case Generation Plan, T&E developed four different taint sources:

- Environment Variable
- File Read
- Socket
- Shared Memory (C only)

In C, these taint sources existed as snippets that were injected into the base program. In Java, these taint sources were created using the Eclipse JDT backend. The code that created the taint sources also provided output functions to be used by the weaknesses.

T&E needed to keep weaknesses from sending output to standard out or standard error, so as not to interfere with the output naturally created by the base program. This was especially important since many output checks for normal base program operation rely on examining the contents of standard out or standard error. Accordingly, T&E developed special output functions for both C and Java that would send output to a known location based on the type of the taint source. For taint sources environment variable, file read, and shared memory, any output from the weakness was sent to a log file at a known location. For socket taint source, however, any output from the weakness was sent back along the socket, and the receiving socket output the returned data to a log file at a known location.

Socket taint source provided an additional complication, because using a socket in this fashion turned any base program into a server. When a base program had a socket taint source, it started a server loop looking for data on that socket. If the data sent to the socket had a specified format, the socket taint source passed the data on to the weakness code. If the data sent to the socket did not have the specified format, the socket taint source echoed the data directly back along the socket. Performer technology was not allowed to simply exit from a server program, regardless of whether the server was intrinsic to the base program (like PostgreSQL or Subversion svnserve) or was inserted as a socket taint source. Exiting from a server program in response to a weakness results in a Denial of Service.

9 Developing Complexity Features

In accordance with the Test Case Generation Plan, T&E developed three different classes of complexity features:

- Control flow
- Data flow
- Data type

These terms are defined in the Test Data Generation Plan, but we summarize the information here.

Control flow refers to the order in which individual instructions are executed. An example of a control flow feature is RECURSIVE, in which a function invokes itself recursively. In this control flow, the data that triggers the weakness is provided before the recursive call, and the vulnerability is triggered inside the recursive call.

Data flow refers to how code passes a source input through the program. An example of a data flow feature is INDEX_ALIAS_1, in which a pointer to the user data is placed into an element of a larger array. The weakness code subsequently extracts the user data from the appropriate index of the array, and uses it to trigger the vulnerability.

Data type refers to how data moves through different types before it is used. An example of a data type feature is VOID_POINTER, in which a pointer to user data is placed into a variable with type void*. The data is later extracted from the void* and used to trigger the vulnerability.

The complexity features developed by T&E were injected directly into the AST using API calls into ROSE (for C) or the Eclipse JDT (for Java). In Phase 2, T&E injected only one complexity feature into each test case, but the injection systems provided the ability to inject multiple complexity features. For Phase 3, T&E settled on adding three complexity features, one from each class: first control flow, then data flow, then data type. This resulted in a large number of possible combined complexity features, as shown in Table 6.

Table 6 Number of Complexity Features

Complexity Feature	C	Java
Control Flow	13	17
Data Flow	17	6
Data Type	7	3
Total Phase 3 Complexity Feature Permutations	13 x 17 x 7 = 1547	17 x 6 x 3 = 306

The Callback control flow feature causes some issues when it interacted with the Socket taint source, the Array data type, and with any data flow. In particular:

- In C, the socket taint source could not be used with the callback control flow feature, because both of them rely on callbacks. If the user requested this combination, the callback control flow feature was replaced with a dummy callback, because a callback already existed in the socket taint source.
- If the user requested a Callback control flow, the callback must occur before any data-flow features and it must also occur before an Array data type feature, in order to pass information correctly to the weakness. If the user requested either of these combinations, the injection system re-ordered the features to place the callback control flow before any data-flow features. In practice, this re-ordering did not occur, since T&E always ordered the complexity features as shown in the table above (control flow, then data flow, then data type). This means that any Callback control flow happened before any data flow or data type, and so there was no need for re-ordering.

For C, there were 1547 possible complexity feature combinations, and since there were 4 taint sources, there were 1547 x 4 = 6188 possible taint source/complexity feature combinations. For Java, there were 306 possible complexity feature combinations and 3 taint sources, resulting in 306 x 3 = 918 possible taint source/complexity feature combinations.

T&E developed unit tests to validate the behavior of the combined complexity features and taint sources, so as to resolve compatibility problems before injecting into base programs.

10 Developing Weakness Snippets

10.1 Initial Weakness Approach

T&E began developing weaknesses for C using the ROSE Compiler Infrastructure. For each weakness, T&E wrote code in C that would execute the weakness. T&E then wrote C++ code that would use the ROSE Compiler Infrastructure to add those C statements to the Abstract Syntax Tree (AST) of the base program. There was a factor of 10 blow-up in lines of code from the C code to the C++ code that generated it. In addition, because the C++ code that actually generated the weakness was one stage removed from the desired C weakness code, it was difficult to maintain and update the weaknesses. Making subsequent changes to the weakness was a slow and error-prone task because of the abstracted nature of the C++ code generating the C code.

10.2 Snippet Weaknesses in Java

T&E initially hoped to use the ROSE Compiler Infrastructure to inject Java weaknesses as well as C weaknesses. However, the Java portion of the ROSE compiler was not as fully developed as the C portion and was buggy, error-prone, slow, and limited in its functionality. Consequently, T&E explored using the Eclipse JDT to inject Java weaknesses. Because Eclipse was a commercial tool with a large installed user base, the interface was much cleaner, and the functionality was more complete.

The Eclipse JDT provided the ability to read in code from two separate files, attach portions of the AST from one file to the AST from the other, and then output the modified AST. This meant that it was possible to write weakness snippet code that would be directly translated into an AST, without a programmer having to hand-construct the AST nodes. This sped up weakness development considerably, and also allowed the weaknesses to be much easier to maintain, understand, and alter.

10.3 Snippet Weaknesses in C

Building on the success of the snippet weakness approach in the Eclipse JDT, T&E tested a snippet weakness functionality provided by the ROSE compiler infrastructure for C. This functionality allowed T&E to write weakness snippets directly in C, have the ROSE compiler turn the snippets directly into ASTs, and then attach those ASTs to the AST for the base program, and output the modified AST.

However, the ROSE C snippet functionality did not have a substantial existing user base, and so the snippet functionality was not as robust as that provided by the Eclipse JDT. Also, the difficulties arising from C preprocessor commands affected the snippet functionality, especially when the snippets used external header files with complicated preprocessor directives. These issues were so pervasive that T&E could not implement all the required snippets using the ROSE compiler snippet functionality.

To resolve these problems while still taking advantage of snippets, T&E then modified the C injection system (ss_vuln_injector) to have a pre-processor driven snippet injection tool. This snippet injection tool read in code from a file and modified necessary variable names by using pre-processor directives. It then attached that code to a node in the ROSE compiler AST using ROSE's ability to attach arbitrary text to a node. This allowed code from the snippet to be inserted into the base program, without having to convert it fully to an AST, thereby bypassing the pre-processor issues. With this solution, T&E was able to implement all C weakness snippets directly in C, without having to hand-construct AST nodes with the ROSE Compiler Infrastructure.

10.4 Snippet Weakness Classes

T&E developed weakness snippets for 6 different weakness classes for C and Java. Each weakness snippet corresponded to a Common Weakness Enumeration (CWE), but there was often more than one algorithmic variant of weakness for a particular CWE. T&E developed the weakness snippets summarized in Table 7.

Table 7 Number of Weakness Snippets

Weakness Class	C		Java	
	CWEs	Snippets	CWEs	Snippets
Concurrency Handling	15	16	15	15
Error Handling	-	-	8	9
Injection	3	8	4	8
Memory Corruption	17	43	-	-
Null Pointer	1	7	-	-
Number Handling	9	11	8	9
Resource Drains	11	13	9	11
Tainted Data	-	-	6	8
Total	**56**	**98**	**50**	**60**

For each weakness snippet, T&E developed benign input values that would not cause any technical impact, and exploit input values that caused some kind of technical impact (denial of service, information leakage, etc.) For C weaknesses, T&E developed 3 benign inputs and 2 exploit inputs. For Java weaknesses, T&E developed either 2 or 3 benign inputs and 2 exploit inputs.

T&E developed unit tests to verify that each weakness could be injected successfully with any possible taint source. These tests verified that the injected code could be compiled and that the YAML was well-formed. However, these tests did not run the injected code or check that inputs were actually benign or exploiting, because such tests would require the significant overhead of a system like TEXAS.

The weaknesses are described in detail in the STONESOUP Test and Evaluation Weakness Documentation. All weakness snippets were validated by the Independent Verification and Validation (IV&V) Team. T&E addressed all issues raised by IV&V regarding the snippets.

10.5 C/Binary Weaknesses

T&E developed the following CWEs for C. Some weaknesses have more than one snippet.

- Number Handling
  - CWE-190: Integer Overflow or Wraparound
  - CWE-191: Integer Underflow (Wrap or Wraparound)
  - CWE-194: Unexpected Sign Extension
  - CWE-195: Signed to Unsigned Conversion Error
  - CWE-196: Unsigned to Signed Conversion Error
  - CWE-197: Numeric Truncation Error
  - CWE-369: Divide By Zero
  - CWE-682: Incorrect Calculation
  - CWE-839: Numeric Range Comparison without Minimum Check.
- Resource Drains
  - CWE-400: Uncontrolled Resource Consumption ('Resource Exhaustion')
  - CWE-401: Failure to Release Memory Before Removing Last Reference ('Memory Leak')
  - CWE-459: Incomplete Cleanup
  - CWE-674: Uncontrolled Recursion
  - CWE-771: Missing Reference to Active Allocated Resource
  - CWE-773: Missing Reference to Active File Descriptor or Handle
  - CWE-774: Allocation of File Descriptors or Handles Without Limits or Throttling
  - CWE-775: Missing Release of File Descriptor or Handle after Effective Lifetime
  - CWE-789: Uncontrolled Memory Allocation
  - CWE-834: Excessive Iteration
  - CWE-835: Infinite Loop
- Injection
  - CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
  - CWE-88: Argument Injection or Modification
  - CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
- Concurrency Handling
  - CWE-363: Race Condition Enabling Link Following
  - CWE-367: Time-of-check Time-of-use (TOCTOU) Race Condition

  - CWE-412: Unrestricted Externally Accessible Lock
  - CWE-414: Missing Lock Check
  - CWE-479: Signal Handler Use of a Non-reentrant Function
  - CWE-543: Use of Singleton Pattern Without Synchronization in a Multithreaded Context
  - CWE-609: Double-Checked Locking
  - CWE-663: Use of a Non-reentrant Function in an Unsynchronized Context
  - CWE-764: Multiple Locks of a Critical Resource
  - CWE-765: Multiple Unlocks of a Critical Resource
  - CWE-820: Missing Synchronization
  - CWE-821: Incorrect Synchronization
  - CWE-828: Signal Handler with Functionality that is not Asynchronous-Safe
  - CWE-831: Signal Handler Function Associated with Multiple Signals
  - CWE-833: Deadlock
- Memory Corruption
  - CWE-120: Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
  - CWE-124: Buffer Underwrite ('Buffer Underflow')
  - CWE-126: Buffer Over-read
  - CWE-127: Buffer Under-read
  - CWE-129: Improper Validation of Array Index
  - CWE-134: Uncontrolled Format String
  - CWE-170: Improper Null Termination
  - CWE-415: Double Free
  - CWE-416: Use After Free
  - CWE-590: Free of Invalid Pointer Not on the Heap
  - CWE-761: Free of Pointer not at Start of Buffer
  - CWE-785: Use of Path Manipulation Function without Maximum-sized Buffer
  - CWE-805: Buffer Access with Incorrect Length Value
  - CWE-806: Buffer Access Using Size of Source Buffer
  - CWE-822: Untrusted Pointer Dereference
  - CWE-824: Access of Uninitialized Pointer
  - CWE-843 Access of Resource Using Incompatible Type ('Type Confusion')
- Null Pointer Errors
  - CWE-476: NULL Pointer Dereference

10.5.1 Weaknesses not implemented in C

Some CWEs were called for in the TGP, but were not implemented in C for one or more of the following reasons:

- Entirely covered by other CWEs
- Not possible to implement this CWE in C
- Not possible to automate testing for this CWE
- Cannot mitigate this CWE without application-specific knowledge

The following CWEs were not implemented in C:

- Resource Drains
  - CWE-404: Improper Resource Shutdown or Release
  - CWE-762: Mismatched Memory Management Routines
  - CWE-770: Allocation of Resources Without Limits or Throttling
- Injection
  - CWE-564: SQL Injection: Hibernate.
- Concurrency Handling
  - CWE-362: Race Condition
  - CWE-364: Signal Handler Race Condition
  - CWE-365: Race Condition in Switch
  - CWE-366: Race Condition within a Thread
  - CWE-558: Use of getlogin() in Multithreaded Application
  - CWE-567: Unsynchronized Access to Shared Data in a Multithreaded Context
  - CWE-572: Call to Thread run() instead of start()
  - CWE-832: Unlock of a Resource that is not Locked
- Memory Corruption
  - CWE-762: Mismatched Memory Management Routines

10.6 Java Weaknesses

T&E developed the following CWEs for Java. Some weaknesses have more than one snippet.

- Number Handling
  - CWE-190: Integer Overflow or Wraparound
  - CWE-191: Integer Underflow (Wrap or Wraparound)
  - CWE-194: Unexpected Sign Extension
  - CWE-195: Signed to Unsigned Conversion Error
  - CWE-196: Unsigned to Signed Conversion Error
  - CWE-197: Numeric Truncation Error
  - CWE-369: Divide By Zero
  - CWE-839: Numeric Range Comparison without Minimum Check

- Tainted Data
  - CWE-15: External Control of System or Configuration Setting
  - CWE-23: Relative Path Traversal
  - CWE-36: Absolute Path Traversal
  - CWE-41: Improper Resolution of Path Equivalence
  - CWE-239: Failure to Handle Incomplete Element
  - CWE-606: Unchecked Input for Loop Condition.
- Error Handling
  - CWE-209: Information Exposure Through an Error Message
  - CWE-248: Uncaught Exception
  - CWE-252: Unchecked Return Value
  - CWE-253: Incorrect Check of Function Return Value
  - CWE-390: Detection of Error Condition Without Action
  - CWE-391: Unchecked Error Condition
  - CWE-460: Improper Cleanup on Thrown Exception
  - CWE-584: Return Inside Finally Block
- Resource Drain
  - CWE-400: Uncontrolled Resource Consumption ('Resource Exhaustion')
  - CWE-459: Incomplete Cleanup
  - CWE-674: Uncontrolled Recursion
  - CWE-773: Missing Reference to Active File Descriptor or Handle
  - CWE-774: Allocation of File Descriptors or Handles Without Limits or Throttling
  - CWE-775: Missing Release of File Descriptor or Handle after Effective Lifetime
  - CWE-789: Uncontrolled Memory Allocation
  - CWE-834: Excessive Iteration
  - CWE-835: Infinite Loop
- Injection
  - CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
  - CWE-88: Argument Injection or Modification
  - CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
  - CWE-564: SQL Injection: Hibernate.
- Concurrency Handling
  - CWE-363: Race Condition Enabling Link Following
  - CWE-367: Time-of-check Time-of-use (TOCTOU) Race Condition
  - CWE-412: Unrestricted Externally Accessible Lock

- CWE-414: Missing Lock Check
- CWE-543: Use of Singleton Pattern Without Synchronization in a Multithreaded Context
- CWE-567: Unsynchronized Access to Shared Data in a Multithreaded Context
- CWE-572: Call to Thread run() instead of start()
- CWE-609: Double-Checked Locking
- CWE-663: Use of a Non-reentrant Function in an Unsynchronized Context
- CWE-764: Multiple Locks of a Critical Resource
- CWE-765: Multiple Unlocks of a Critical Resource
- CWE-820: Missing Synchronization
- CWE-821: Incorrect Synchronization
- CWE-832: Unlock of a Resource that is not Locked
- CWE-833: Deadlock

10.6.1 Weaknesses Not Implemented in Java

Some CWEs were called for in the TGP, but were not implemented in Java for one or more of the following reasons:

- Entirely covered by other CWEs
- Not possible to implement this CWE in Java
- Not possible to automate testing for this CWE
- Cannot mitigate this CWE without application-specific knowledge

The following CWEs were not implemented in Java:

- Number Handling
  - CWE-682: Incorrect Calculation
- Error Handling
  - CWE-273: Improper Check for Dropped Privileges
  - CWE-274: Improper Handling of Insufficient Privileges
  - CWE-280: Improper Handling of Insufficient Permissions or Privileges
  - CWE-394: Unexpected Status Code or Return Value
  - CWE-395: Use of Null Pointer Exception Catch to Detect NULL Pointer Dereference
  - CWE-396: Declaration of Catch for Generic Exception
  - CWE-397: Declaration of Throws for Generic Exception
  - CWE-600: Failure to Catch All Exceptions in Servlet
  - CWE-617: Reachable Assertion.
  - CWE-698 Redirect without Exit.
- Resource Drain
  - CWE-401: Failure to Release Memory Before Removing Last Reference ('Memory Leak')

  - CWE-404: Improper Resource Shutdown or Release
  - CWE-762: Mismatched Memory Management Routines
  - CWE-770: Allocation of Resources Without Limits or Throttling
  - CWE-771: Missing Reference to Active Allocated Resource
- Concurrency Handling
  - CWE-362: Race Condition
  - CWE-364: Signal Handler Race Condition
  - CWE-365: Race Condition in Switch
  - CWE-366: Race Condition within a Thread
  - CWE-479: Signal Handler Use of a Non-reentrant Function
  - CWE-558: Use of getlogin() in Multithreaded Application
  - CWE-828: Signal Handler with Functionality that is not Asynchronous-Safe
  - CWE-831: Signal Handler Function Associated with Multiple Signals

11 Trace data

To aid in debugging weaknesses and in evaluating the performance characteristics of the performer technology, T&E added trace statements using the Linux Trace Toolkit Next Generation (LTTng), available at https://lttng.org/. This was a lightweight tracing solution that allows tracing to be activated or not at run-time, and that outputs time stamps and specified data (include memory addresses and return pointer contents) to a trace file for later analysis.

T&E added trace statements at the beginning and end of the program, at the beginning and end of each weakness, and at points throughout the weakness. Within the weakness, T&E added trace statements before and after the crossover point, which was the point at which the system enters an unintended state, and before and after the trigger point, which was the point at which the system experiences a negative technical impact. It was not always possible to place a trace statement after the trigger point. For instance, if the trigger happened upon return from the weakness stack frame, there was no way to inject a trace statement after that return occurred, because that code location was not available at compile-time. In addition, if the weakness triggered a system crash, any trace statement placed after the trigger would not be executed.

T&E additionally placed trace statements at locations that would help a reviewer understand the behavior of the weakness code, including at the beginning of most function bodies within the weakness. Trace statements were omitted from function bodies that were called hundreds of times (or more), to keep the size of the trace logs manageable.

The trace statements within the weakness allowed T&E to verify test cases were running as expected. This was especially useful for test cases that were multi-threaded, such as concurrency-handling weaknesses. It was also useful for post-hoc analysis of failing test cases.

For more details on LTTng trace data, and its use in performance analysis, see the STONESOUP Phase 3 Test and Evaluation Final Report.

12 Corpus Specification Generation

The automated injection system allowed the generation of a large number of test cases to use in evaluating performer technology. To construct a test case, the user must select values for the following:

- Weakness and algorithmic variant
- Base program
- Injection point
- Taint source
- Control flow complexity feature
- Data flow complexity feature
- Data type complexity feature

Weakness classes have different numbers of algorithmic variants, so the number of different possible test cases is different for each weakness class.

Table 8 indicates the number of choices for each of the parameters except weakness and algorithmic variant in C and in Java.

Table 8 Number of choices for test case parameters, except weakness

Test Case Parameter	C/Binary	Java
Base Program	6	6
Injection Points	10	10
Taint Source	4	3
Control Flow Complexity Features	10	12
Data Flow Complexity Features	11	6
Data Type Complexity Features	7	3

These parameters could all be set independently, so for each weakness algorithmic variant, the number of possible combinations of other parameters is:

- 6 x 10 x 4 x 10 x 11 x 7 = 184,800 combinations for C/Binary
- 6 x 10 x 3 x 12 x 6 x 3 = 38,800 combinations for Java

Table 9 shows the number of snippets and the number of possible test cases for each weakness class.

Table 9 Possible test cases for each weakness class

Weakness Class	C/Binary		Java	
	Snippets	Possible Test Cases	Snippets	Possible Test Cases
Concurrency Handling	16	2,956,800	15	582,000
Error Handling	-	-	9	349.200
Injection	8	1,478,400	8	310,400
Memory Corruption	43	7,946,400	-	-
Null Pointer	7	1,293,600	-	-
Number Handling	11	2,032,800	9	349.200
Resource Drains	13	2,402,400	11	426,800
Tainted Data	-	-	8	310,400

Because of constraints of time and processing power, it was not possible to run every single one of these test cases. Accordingly, T&E developed a test corpus consisting of a subset of these test cases.

For each performer, T&E selected test cases at random from each appropriate weakness class, subject to constraints on the outcome distribution. Table 10 shows the number of test cases selected from each weakness class for each performer.

Table 10 Number of test cases per weakness class

Performer	Language	Number of test cases chosen per weakness class
Columbia	C	531
Grammatech	Binary	637
Kestrel	Java	478

To develop the corpus specification, T&E wrote a program that generated possible test case names and then validated each test case name for uniformity on various axes. Uniformity was defined as the lowest possible delta between the counts of each of the possible values.

The algorithm selected values for axes in the following order:

- Language
- Weakness class
- Weakness and algorithmic variant within weakness class
- Base program
- Injection point within base program
- Taint source
- Control flow
- Data flow
- Data type

Figure 7 illustrates the process used to generate a test corpus specification. For each axis in order, the algorithm picked a value for that axis that had been used the least in the previously accepted test case names. It then checked that the new value, in combination with the values chosen for the previous axes, passed all required uniformity checks. If any uniformity check failed, the algorithm chose a different least-used value for the axis and tried again. If no choice of value for the axis satisfied all uniformity checks, the algorithm recursively fell back to the previous axis, and tried again.

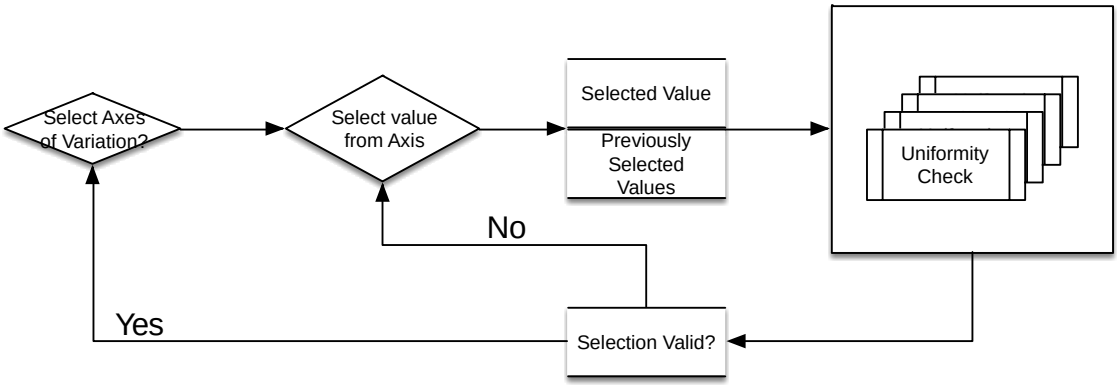

Figure 7 Process for Test Corpus Generation Algorithm

The test corpus developed by T&E was as uniform as possible on several specified subsets of the test case parameters. In particular, T&E tested for uniformity on the following individual parameters:

- Weakness Class
- Weakness within Weakness Class
- Base Program
- Injection Point
- Taint Source
- Control Flow
- Data Flow
- Data Type

In addition, T&E tested for uniformity on the following combinations of parameters:

- Base Program/Injection Point
- Base Program/Taint Source
- Base Program/Control Flow
- Base Program/Data Flow
- Base Program/Data Type
- Taint Source/Control Flow
- Taint Source/Data Flow

- Taint Source/Data Type
- Data Flow/Control Flow
- Data Type/Control Flow
- Data Type/Data Flow
- Weakness/Base Program
- Weakness/Taint Source
- Weakness/Control Flow
- Weakness/Data Flow
- Weakness/Data Type
- Taint Source/Data Type/Control Flow
- Taint Source/Data Type/Data Flow
- Taint Source/Data Flow/Control Flow
- Taint Source/Data Type/Data Flow/Control Flow
- Weakness/Base Program/Injection Point

Each of these checks was required to be separately as uniform as possible with the entire collection of checks providing optimum uniformity.

A full discussion of corpus specification generation, including the tests used to measure uniformity and the statistical metrics of the final specification corpus is included in the STONESOUP Phase 3 Test and Evaluation Final Report.

13 Packaging Test Cases

The packager was contained in the ss_testcases repository. This tool combined all the necessary components into a test case that could be run by TEXAS. Figure 2 in Section 2.4 shows the inputs and outputs for the packager.

13.1 Inputs to Packager

The packager took as input the following items:

- Test case name
- Base program information
- Weakness information

13.1.1 Test Case Name

The test case name provided the following information:

- Base program
- Injection point for the base program
- Taint source type
- Control flow complexity feature
- Data flow complexity feature
- Data type complexity feature

13.1.2 Base Program

The packager required the following information for the base program:

- Source code, modified to respect library environment variables and to be able to be processed by the injection system
- Skeleton, which was the directory structure used by base program inputs
- XML containing:
    - Instructions for building base program
    - For each of the 10 different inputs:
        - Instructions for running base program for this input
        - Pre-processes required to run before the base program was run, e.g.:
            - Setting environment variables
            - Creating files with particular contents in particular directories
            - Running scripts to make sure that the system was in the correct state for the test case
        - Co-processes required to run at the same time the base program was run, e.g:
            - Running a client program for a server program
            - Running a script to provide automated GUI input
        - Post-processes required to run after the base program was run, e.g.:

        - Shutting down a server operating during the test case
    - Output checks to verify that the input ran as expected. These could be arbitrarily nested Boolean expressions, and could include such tests as:
        - Looking in a particular location for a particular string
        - Looking in a particular location for a specified regular expression
        - Checking that a particular file was created with particular contents
        - Checking that the test case did (or did not) time out.

The pre-, co-, and post-processes were a general interface allowing any script to be run to support the test case. The processes could be run sequentially or in parallel, and a collection of processes (run sequentially or in parallel) could be nested arbitrarily.

The packager required the following information for the weakness snippet:

- Snippet source code
- Snippet YAML, containing:
    - 2 or 3 benign inputs that would exercise the weakness without having a technical impact.
    - 2 exploit inputs that would cause some kind of technical impact, such as denial of service or information exposure

For each benign or exploit input, the snippet YAML file provided the value that would be provided to the program through the taint source. In addition, for each benign or exploit input, the snippet YAML file could specify other pre-, co-, or post-processes that needed to run to support the snippet. These processes were a generalized structure to run any kind of script. They could do things like set up particular environment variables or files, run co-processes to coordinate actions of concurrency weaknesses, or run post-processes to clean up large files generated by the weakness.

The snippet YAML file did not contain output checks or technical impacts for the exploit inputs. Due to schedule constraints, the technical impact and output checks for exploit inputs were contained in the code of the packager. Given more programming resources, it would make sense to move these checks to the snippet YAML files themselves, but this would require a significant effort.

13.2 Outputs from Packager

The packager produced a test case tarball suitable for running in TEXAS. This tarball contains:

- The injected base program
- The injected skeleton
- The injected YAML
- The inject XML

13.2.1 Injected Base Program

The injected base program was created when the injection system (ss_vuln_injector for C/Binary or ss_vinject4j for Java) processed a base program. The injection system began by reading in a base program that had been appropriately modified by T&E to respect library environment variables and to allow processing by the injection system. The injection system inserted the atomic barrier, and within that inserted the taint source, complexity features, and weakness specified by the test case name. The injection system then output the source code for use in the test case tarball.

13.2.2 Injected Skeleton

The packager created the injected skeleton beginning with the base program skeleton, which provided necessary files for running the base program. It then added to that skeleton files that were required for the chosen weakness. For instance, if the test case name specified the socket taint, then the skeleton would need to have access to the service_mon.sh script, which checked whether a service was up or not. This script was placed into the skeleton's scripts directory, and instructions were added to the injected XML file to run this script as a co-process. Similarly, if the test case name specified the shared memory taint, then the skeleton would need scripts to establish and tear down the shared memory, and the injected XML file would need instructions to run those scripts when appropriate. If the weakness required a common script such as the runFifos.py script used by concurrency weaknesses, that script would also be downloaded into the skeleton's scripts folder.

13.2.3 Injected YAML

The packager created the injected YAML beginning with the YAML for the chosen weakness. It added details about the injection point, the taint source, and the code complexity features used in the test case. It provided input details for 2 or 3 inputs that result in benign behavior by the weakness, and 2 inputs that result in exploit behavior by the weakness.

13.2.4 Injected XML

The packager created the injected XML from the base program XML and the injected YAML. The base program XML provided the necessary instructions for building and running the base program. The injected YAML provided the necessary inputs for running the weakness, either in a benign fashion or an exploit fashion.

The injected XML file had 10 good IO pairs and 2 bad IO pairs. Each IO pair had:

- Inputs for the base program, and
- Inputs for the weakness

The 10 good IO pairs each had:

- One of the 10 possible inputs for the base program, selected in order
- One of the 2 or 3 benign inputs for the weakness, chosen by cycling deterministically through the available benign inputs

The 2 bad IO pairs each had:

- One of the 10 possible inputs for the base program, selected at random with replacement
- One of the 2 exploit inputs for the weakness, selected in order

In constructing the injected XML for each IO pair, the packager created:

- Collection of pre-processes from:
    - Pre-processes for the appropriate base program input
    - Pre-processes for appropriate taint source
    - Pre-processes for appropriate weakness input
- Run-command - Uses the run command for the appropriate base program input
- Collection of co-processes from:
    - Co-processes for the appropriate base program input
    - Co-processes for appropriate taint source
    - Co-processes for appropriate weakness input
- Collection of post-processes from:
    - Post-processes for the appropriate base program input
    - Post-processes for appropriate taint source
    - Post-processes for appropriate weakness input
- Set of output checks:
    - For good IO pairs, used the output check for the appropriate input from the base program. These output checks were specified in the base program XML file

    - For bad IO pairs, used the output check for the appropriate exploit input from the weakness. These output checks were specified in the packager source code.
- Technical impact of weakness – NONE for benign inputs, and an explanation of the technical impact for exploit inputs. The technical impact for each weakness exploit input was specified in the packager source code.

The collections of pre-, co-, and post-processes were each an ordered group of actions. Processes could be run either sequentially or in parallel, and a process could consist of a sub-collection of processes that were themselves run either sequentially or in parallel. Collections of processes could be nested arbitrarily.

The output checks consisted of a Boolean formula (with arbitrary nesting) of checks on return codes, checks for particular strings in particular locations, checks of script outputs, and checks of timeout (or not).

14 Debugging Test Cases in Base Programs

Upon packaging test cases, T&E ran them through TEXAS to validate them. IV&V validated which test cases ran successfully through TEXAS under Stage One (without performer technology) and which had errors.

TEXAS took as input a tarball for the test case. It built the test case, using the instructions in the XML file to generate an analyze tarball. Then it ran the test case on each of the 12 IO pairs (10 good and 2 bad), and generated an execute tarball for each IO pair.

If a test case failed, T&E debugged the problem using the results of the analyze or execute tarball. This tarball contained:

- The injected base program, with any modifications made by the build or run process
- The injected skeleton, with any modifications made by the run process, including information written by the weakness to the designated log file
- The injected YAML
- The injected XML
- Log data including:
    - LTTng trace data
    - Standard out and standard error for the build commands, and for each pre-process, co-process, post-process, and run command

To debug a test case, T&E generally began by examining the log files created by the various processes, and proceeded to look at the modifications made to the base program and the skeleton. From there, T&E re-constructed the actions taken by the analyze or execute run to find the problem.

15 Lessons Learned

Automated test generation is a difficult task. T&E notes the following lessons from this project:

- Snippet functionality was absolutely essentially to writing and maintaining a large library of injectable code. It was extremely difficult to maintain or update code that generated other code. It was not a scalable process to write code that generated AST nodes that were then output as code.
- GUI automation was a difficult and time-consuming task. GUIs can utilize a wide variety of appearances and actions, because the entity interacting with them is usually a human being with robust error-correcting capability. GUIs are not designed to interact with automated systems that do not have such robust error-correcting capability. In addition GUIs may take an alternate path only a small percentage of the time, making recognizing and debugging issues more difficult still. Significant time must be allotted for multiple runs of GUI applications to observe and remove all bugs.
- Dynamically loaded libraries would make it easier to add code to base programs. Having to alter a base program's build processes to respect library environment variables requires a significant input of programmer time. It may be more efficient to use dynamically loaded libraries to access outside functionality, though such dynamic loading may create difficulties for performer technology.
- Multi-level output checks were difficult to get right, because of their abstract nature and obtuse syntax. They require hand-checking and careful thought to iron out problems.

Appendix A: Acronyms

Table 11 Acronym List

Acronym	Acronym Definition
API	Application Programming Interface
AST	Abstract Syntax Tree
CWE	MITRE Common Weakness Enumeration
Eclipse JDT	Eclipse Java Development Toolkit
GUI	Graphical User Interface
IARPA	Intelligence Advanced Research Projects Activity
IV&V	STONESOUP Independent Verification and Validation Team
JVM	Java Virtual Machine
LTTng	Linux Trace Toolkit Next Generation
STONESOUP	Securely Taking On New Executable Software Of Uncertain Provenance
T&E	STONESOUP Phase 3 Test and Evaluation Team
TGP	STONESOUP Phase 3 Test Case Generation Plan
TEXAS	STONESOUP Phase 3 Test and Evaluation eXecution and Analysis System
XML	eXtensible Markup Language
YAML	YAML Ain’t Markup Language