


Juliet Test Suite v1.2 for C/C++ User Guide



Center for Assured Software
National Security Agency
9800 Savage Road
Fort George G. Meade, MD 20755-6738
cas@nsa.gov

December 2012

Table of Contents

Section 1: Introduction.....	1
1.1 Document Purpose	1
1.2 What are Test Cases?	1
1.3 Why Test Cases?.....	1
1.3.1 Limitations of Natural Code	1
1.3.2 Limitations of the Test Cases.....	2
1.4 Creating Test Cases.....	3
1.5 Feedback	3
Section 2: Test Case Scope	4
2.1 Test Case Selection	4
2.2 Test Case Statistics	5
Section 3: Test Case Naming.....	7
3.1 Naming Scheme	7
3.2 Test Case Functional Variants	7
3.2.1 Key Strings in Functional Variant Names	7
3.3 Test Case Flow Variants	8
3.4 Test Case Files	9
3.4.1 Test Case File Names.....	10
3.4.2 Sub-file Identifier	11
Section 4: Test Case Design	12
4.1 Non-Class-Based Flaw Test Cases	12
4.1.1 Required Functions	12
4.1.2 Optional Functions.....	14
4.2 Class-Based Flaw Test Cases.....	16
4.3 Virtual Function Test Cases.....	18
4.4 Constructor/Destructor Test Cases	18
4.5 Bad-only Test Cases	19
Section 5: Test Case Support Files	20
5.1 Common Support Files	20
5.2 CWE Entry Specific Support Files	20
5.3 Main Support Files.....	21
Section 6: Building Test Cases	23
6.1 Build Prerequisites	23
6.2 Compiling Test Cases	23
6.2.1 Compile All Test Cases as One Executable.....	23
6.2.2 Compile Test Cases per CWE Entry.....	24
6.3 Compiling an Individual Test Case.....	24
6.3.1 Building and Running a Test Case.....	25
Section 7: Updating Build Files.....	26
7.1 Updating C/C++ Build Files	26
Section 8: Tool Analysis.....	27
8.1 True Positives and False Negatives	27
8.2 False Positives and True Negatives	27

8.3 Unrelated Flaw Reports	28
Appendix A: Test Case CWE Entries.....	A-1
Appendix B: CWE/SANS Top 25 Coverage	B-1
Appendix C: Test Case Flow Variants	C-1
Appendix D: Bad-Only Test Cases	D-1
Appendix E: Test Case Changes in v1.2.....	E-1

Section 1: Introduction

1.1 Document Purpose

This document describes the Juliet Test Suite v1.2 for C/C++. The test suite was created by the National Security Agency's (NSA) Center for Assured Software (CAS) and developed specifically for assessing the capabilities of static analysis tools. It is intended for anyone who wishes to use the test cases for their own testing purposes, or who would like to have a greater understanding of how the test cases were created.

This document explains the philosophy behind the naming and design of the test cases and provides instruction on how to compile and run them using a Command Line Interface (CLI). Section 8 also provides details on how the tool results can be evaluated.

The test cases are publically available for download at <http://samate.nist.gov/SRD/testsuite.php>.

1.2 What are Test Cases?

Test cases are pieces of buildable code that can be used to study static analysis tools. A test case targets exactly one type of flaw, but other, unrelated flaws may be incidentally present. For example, the C test case "CWE476_NULL_Pointer_Dereference__char_01" targets only a NULL Pointer Dereference flaw. In addition to the construct containing the target flaw, each test case typically contains one or more non-flawed constructs that perform a function similar to the flawed construct. A small subset of test cases does not contain non-flawed constructs and are considered "bad-only" test cases (see Section 4.5).

1.3 Why Test Cases?

In order to study static analysis tools, the CAS needs software for tool analysis. The CAS previously considered using "natural" or "artificial" software. Natural software is software that was not created to test static analysis tools. Open source software applications, such as the Apache web server (<http://httpd.apache.org>) and the OpenSSH suite (www.openssh.com), are examples of natural software. Artificial software, in this case, is software that contains intentional flaws and is created specifically to test static analysis tools. The test cases are an example of artificial software.

1.3.1 Limitations of Natural Code

During previous research efforts, the CAS used a combination of natural and artificial code in testing static analysis tools. In addition, the CAS followed the National Institute of Standards and Technology (NIST) Static Analysis Tool Exposition (SATE) that examined the performance of static analysis tools on natural code.

Experiences from these efforts indicated that the use of natural code often presents specific challenges, such as:

- Evaluating tool results to determine their correctness – When a static analysis tool is run on natural code, each result needs to be reviewed to determine if the code in fact has the specified type of flaw at the specified location (i.e. if the result is correct or a “False Positive”). This review is non-trivial for most results on natural code and often the correctness of a given result cannot be determined with a high degree of certainty in a reasonable amount of time.
- Comparing results from different tools – Comparing static analysis tool results on natural code is complicated because different tools report results in different manners. For example, many flaws involve a “source” of tainted data and a “sink” where that data is used inappropriately. Some tools may report the source where others report the sink. Sometimes multiple sources of tainted data all lead to one sink, which may cause different tools to report a different number of results.
- Identifying flaws in the code that no tools find – When evaluating static analysis tools, a “standard” list of all flaws in the code is needed in order to identify which flaws each tool failed to report. On natural code, creating this “standard” is difficult, especially identifying flaws that are not reported by any automated tool and therefore can only be found with manual code review.
- Evaluating tool performance on constructs that do not appear in the code – Natural code has the limitation that even a combination of different projects will likely not contain all flawed and non-flawed constructs that the CAS wants to test. Even flaw types that appear in the code may be obfuscated by complex control and data flows such that a flaw in the natural code will remain undetected even by tools that generally catch a flaw of that type. To address this issue, the CAS considered using a “seeding” method to embed flaws and non-flaws into natural code. Ultimately, test cases were created instead of using “seeding” because the CAS believed that studying static analysis tools using “seeded” code would be overly complex and result in testing fewer constructs than desired.

Based on these experiences and challenges, the CAS decided to develop artificial test cases to test static analysis tools. Using artificial code simplifies tool studies by allowing the CAS to control, identify, and locate the flaws and non-flaws included in the code.

1.3.2 Limitations of the Test Cases

Although the use of the test cases simplifies static analysis tool studies, it may limit the applicability of results in the following two ways:

- Test cases are simpler than natural code – Some test cases are intentionally the simplest form of the flaw being tested. Even test cases which include control or data flow complexity are relatively simple compared to natural code, both in terms of the number of lines of code and in terms of the number and types of branches, loops, and function

calls. This simplicity may inflate results in that tools may report flaws in the test cases that they would rarely report in natural, non-trivial code.

- Frequencies of flaws and non-flawed constructs in the test cases may not reflect their frequencies in natural code – Each type of flaw is tested once in the test cases, regardless of how common or rare that flaw type may be in natural code. For this reason, two tools that have similar results on the test cases may provide very different results on natural code, such as if one tool finds common flaws and the other tool only finds rare flaws. Even a tool with poor results on the test cases may have good results on natural code. Similarly, each non-flawed construct also appears only once in the test cases, regardless of how common the construct is in natural code. Therefore, the False Positive rates on the test cases may be much different from the rates the tools would have on natural code.

1.4 Creating Test Cases

Most of the test cases for non-class-based flaws were generated using source files that contain the flaw and a tool called the “Test Case Template Engine” created by the CAS. Generated test case files contain a comment in the first line indicating that they were generated.

Some flaw types could not be generated by the CAS’s custom Test Case Template Engine. Test cases for those flaw types were manually created. Due to resource constraints, these test cases were created to include only the simplest form of the flaw without added control or data flow complexity.

1.5 Feedback

If you have questions, comments, or suggestions on how to improve the test cases, please contact the CAS via e-mail at CAS@nsa.gov.

Section 2: Test Case Scope

This section provides details on the scope of the test cases. In general, the test cases are focused on functions available on the underlying platform rather than the use of third-party libraries.

Although C and C++ are different programming languages, they are treated as a single unit since C++ is generally a superset of C. In addition, most software assurance tools support both C and C++.

Wherever possible, the C/C++ test cases restrict Application Programming Interface (API) calls to the C standard library, which is available on all platforms. In order to cover more issues, some test cases target the Windows platform (using Windows-specific API functions). In the future, this effort could be expanded to cover API functions that are unique to platforms other than Windows. No third-party C or C++ library functions are used.

The C test case code targets the C89 standard so that the test cases can be compiled and analyzed using a wide variety of tools that may not support newer versions of the C language.

The test cases limit the use of C++ constructs and features to only the test cases that require them (such as test cases related to C++ classes or the “new” operator). Unless necessary for the flaw type targeted, test cases do not use the C++ standard library.

2.1 Test Case Selection

The CAS uses several sources when selecting flaw types for test cases:

- The test case development team’s experiences in Software Assurance
- Flaw types used in the CAS’s previous tool studies
- Vendor information regarding the types of flaws their tools identify
- Weakness information in MITRE’s Common Weakness Enumeration (CWE)¹

While each test case uses a CWE identifier as part of its name, a specific CWE entry for a flaw type is not required in order to create a test case. Test cases are created for all appropriate flaw types and each one is named using the most relevant CWE entry (which might be rather generic and/or abstract).

¹ The MITRE CWE is a community-developed dictionary of software weakness types and can be found at: <http://cwe.mitre.org>

2.2 Test Case Statistics

The test cases cover 11 of the 2011 CWE/SANS Top 25 Most Dangerous Software Errors. Of the 14 CWE entries in the Top 25 that the test cases do not cover, 10 are design issues that do not fit into the structure of the CAS test cases. The other four are not specific to C/C++ and are covered in the related Java test cases. (See Appendix B for details on the test cases associated with each of the Top 25).

New flaws were added in the Juliet Test Suite v1.2 for C/C++. The number of C/C++ test cases in 2012 totaled 61,387, as opposed to 57,099 in 2011. This represents an increase of 7.5%. Table 1 contains statistics on the size and scope of the test cases for 2011 and 2012.

	2011	2012	Percentage Change
CWE Entries Covered	119	118	-0.8%
Flaw Types	1,489	1,617	8.6%
Test Cases	57,099	61,387	7.5%
Lines of Code²	8,375,604	8,679,682	3.6%

Table 1 – 2011-2012 C/C++ Test Case Statistics

See Appendix A for a complete list of the CWE entries covered by the test cases.

² Counted using CLOC (cloc.sourceforge.net). Blank or commented lines were not included. Includes main functions.

In addition, the following changes occurred in Juliet Test Suite v1.2 for C/C++:

- Test cases for an additional six CWEs were added.
- Test cases for seven CWEs were removed.
- One flow variant was removed.
- Two flow variants were added.
- The number of flaw types for 22 CWEs either increased or decreased.
- Some test case directories were split into smaller subdirectories so that each one contains no more than 1,000 test case files.
- Removal of dead code from several control flow variants.

See Appendix E for more details.

Section 3: Test Case Naming

As described in Section 1.2, test cases are pieces of buildable code that target exactly one type of flaw and typically contain one or more non-flawed constructs that perform a function similar to the flawed construct.

3.1 Naming Scheme

The test cases use MITRE's CWEs as a basis for naming and organization. The test cases strive to use the most specific CWE entry for the target flaw. Each test case file is associated with exactly one CWE entry.

A test case is uniquely identified by a combination of four elements:

- The identifying number and possibly shortened name of the CWE entry most closely associated with the intentional flaw.
- A “functional variant” name, which indicates the intentional flaw more specifically than the CWE entry.
- A two-digit number associated with a “flow variant” which indicates the type of data and/or control flow used in the test case. For example, flow variant “01” is the simplest form of the flaw and contains neither data nor control flows.
- The programming language used in the test case. This is indicated in the extension(s) for the test case files (“.c,” “.cpp,” or “.h”).

The name for a test case is written as “C test case CWE476_NULL_Pointer_Dereference__char_01.” Single file test cases can also be referenced by the file name.

3.2 Test Case Functional Variants

Every test case has a “functional variant” name. The term functional variant is also synonymous with “flaw type.” This word or phrase is used to differentiate test cases for the same CWE entry. It should be as short as possible and will often be simply the name of a type or function used in the test case. If there is only one type of issue for a CWE entry, then the functional variant name for test cases for that CWE entry is “basic.”

3.2.1 Key Strings in Functional Variant Names

There is a key string that can appear in functional variant names to indicate test case characteristics. This string is used by scripts that manage the test cases, build process, and result evaluation. Due to the nature of the software used to generate most test cases, this string may appear more than once in a functional variant name:

- “w32” – This string in the functional variant name for a test case indicates that the functional variant is specific to the Windows operating system. Typically, these test cases use functions in the “win32” API that are not present in other operating systems. Many of the C/C++ test cases will compile on non-Windows platforms, however these will not. An example of such a test case is the C test case `CWE78_OS_Command_Injection__char_listen_socket_w32_execv_41.c`.

3.3 Test Case Flow Variants

The test cases are used to demonstrate the ability of static analysis tools to follow various control and data flows in order to properly report a flaw and properly disregard a non-flaw in software. The type of control or data flow present in a test case is specified by the “flow variant” number. Test cases with the same flow variant number (but a different CWE entry or “functional variant”) are using the same type of control or data flow.

Test cases with a flow variant of “01” are the simplest form of the flaws and do not contain added control or data flow complexity. This set of test cases is referred to as the “Baseline” test cases.

Test cases with a flow variant other than “01” are referred to as the “More Complex” test cases. Those with a flow variant from “02” to “22” (inclusive) cover various types of control flow constructs and are referred to as the “Control Flow” test cases. Those with a flow variant of “31” or greater cover various types of data flow constructs and are referred to as the “Data Flow” test cases. The gap between 22 and 31 is left to allow for future expansion.

Some flaw types do not have test cases for every flow variant. There are several reasons for this as not all of the flaw types:

- Involve “data” and therefore cannot be used in Data Flow test cases.
- Can be placed in Control or Data flows because the flaw is inherent in a C++ class (only a Baseline test case is possible for these flaw types).
- Can be generated by the current version of the CAS’s custom Test Case Template Engine and as a result are manually created. Only Baseline (“01” flow variant) test cases are created for these flaw types. In the future, more complex test cases may be created for these flaw types, either manually or through the use of an enhanced version of the engine.
- Support compatibility with all of the control and data flows and may result in a test case that will not compile or function appropriately. Some of these issues are unavoidable because the problem is inherent in the combination of the flaw type and the flow variant. Other compatibility issues involve limitations of the current Test Case Template Engine. Future versions of the test engine may contain additional combinations.

The flow variants used in the test cases are detailed in Appendix C.

3.4 Test Case Files

A test case file is a file that is associated with exactly one test case (as opposed to test case supporting files that are typically used by multiple test cases). An individual test case consists of one or more test case file(s). Below are examples of test cases and their associated file names:

C test case `CWE476_NULL_Pointer_Dereference__char_01` consists of one file:

- `CWE476_NULL_Pointer_Dereference__char_01.c`

C test case `CWE476_NULL_Pointer_Dereference__char_51` consists of two files:

- `CWE476_NULL_Pointer_Dereference__char_51a.c`
- `CWE476_NULL_Pointer_Dereference__char_51b.c`

C test case `CWE476_NULL_Pointer_Dereference__char_54` consists of five files:

- `CWE476_NULL_Pointer_Dereference__char_54a.c`
- `CWE476_NULL_Pointer_Dereference__char_54b.c`
- `CWE476_NULL_Pointer_Dereference__char_54c.c`
- `CWE476_NULL_Pointer_Dereference__char_54d.c`
- `CWE476_NULL_Pointer_Dereference__char_54e.c`

C++ test case `CWE563_Unused_Variable__unused_class_member_value_01` consists of two files:

- `CWE563_Unused_Variable__unused_class_member_value_01_bad.cpp`
- `CWE563_Unused_Variable__unused_class_member_value_01_good1.cpp`

Test cases are not entirely self-contained. They rely on other files called test case support files, which are described in Section 5.

3.4.1 Test Case File Names

Test case files are named with the following parts in order:

Part	Description	Optional/Mandatory
"CWE"	String Literal	Mandatory
CWE ID	Numerical identifier for the CWE entry associated with this test case, such as "36"	Mandatory
" "	String Literal	Mandatory
Shortened CWE entry name	A potentially shortened version of the CWE entry name, with underscores between words, such as "Absolute_Path_Traversal"	Mandatory
"_" (two underscores)	String Literal	Mandatory
Functional Variant Name	A word or short phrase describing this particular variant of the issue, such as "fromConsole." This item is described further in Section 3.2 above.	Mandatory
" "	String Literal	Mandatory
Flow Variant	A two digit integer value describing the type of complexity of the test case, such as "01," "02," or "61." This item is described further in Section 3.3 above.	Mandatory
Sub-file Identifier	A string that identifies this file in a test case consisting of multiple files, such as "a," "b," "_bad," "_good1." This item is described further in Section 3.4.2 below.	Optional
"."	String Literal	Mandatory
Language identifier / file extension	String Literal "c," "cpp," or ".h"	Mandatory

Table 2 – Test Case File Name Components

For example, consider a test case written to evaluate a tool's ability to find integer overflows. This test case reads input from the console using the "fscanf" function and adds two numbers. This test case is the simplest form of this flaw and is contained in one file:

CWE Entry ID: 190
Shortened CWE Entry Name: "Integer_Overflow"
Functional Variant: "char_fscanf_add"
Flow Variant: 01
Language: C

The test case will be contained in the file named:

CWE190_Integer_Overflow__char_fscanf_add_01.c

3.4.2 Sub-file Identifier

The simpler forms of most flaws can be contained in a single source code file, but some test cases consist of multiple files. There are several reasons a test case may be split into multiple files and each one uses a different type of string to identify each file in the test case.

- Some C++ flaws are inherent in a class and require separate files for the flawed and non-flawed constructs. In this case, the flaw will be in a file identified with the string “_bad” (such as “CWE401_Memory_Leak__destructor_01_bad.cpp”) and the non-flaw will be in the file identified with the string “_good1” (such as “CWE401_Memory_Leak__destructor_01_good1.cpp”). Section 4.2 contains more information about class-based flaws.
- Some Data Flow test cases involve the flow of data between functions in different source code files. In these test cases, the test case will “start” in the file identified with the string “a,” such as “CWE476_NULL_Pointer_Dereference__char_54a.c.” Functions in the “a” file will call functions in the “b” file, which may call functions in the “c” file, etc.
- Some Data Flow test cases involve the flow of data between virtual function calls. In the C++ version of these test cases, a header file (.h) is used to define the virtual function and implementations occur in separate source (.cpp) files.
- Some Data Flow test cases involve the flow of data between a class constructor and destructor. In these test cases, a header file (.h) is used to define the constructor and destructor and the implementations occur in separate source (.cpp) files.

Section 4: Test Case Design

Most test cases cover flaws that can be contained in arbitrary functions (non-class-based flaws). However, some flaws, called class-based flaws, are inherent in the C++ class definition and must be handled differently in the test case design. An example of a class-based flaw is:

C++ test case CWE416_Use_After_Free__operator_equals_01

(In this test case, failure to define operator= may result in a program crash due to potentially using memory after it has been freed.)

Virtual function, constructor/destructor, and bad-only test cases are unique. Virtual function and constructor/destructor test cases require multiple files while bad-only test cases are only used to test flaws, as opposed to testing both flaws and non-flaws as in all other test cases.

All C/C++ test cases also define a “main” function in the primary file. This main function is not used when multiple test cases are compiled at once. However, it can be used when building an individual test case, such as for developer testing or for creating binaries to use in testing binary analysis tools.

In the C/C++ test cases, the preprocessor macro INCLUDEMAIN must be defined at compile time for this main function to be included in the compilation.

The sections below describe the test case design for non-class-based flaw, class-based flaw, virtual function, constructor/destructor and bad-only test cases.

4.1 Non-Class-Based Flaw Test Cases

4.1.1 Required Functions

Test cases for flaws that are not inherent in a C++ class must define bad and good functions. (Note: A few test cases are considered bad-only and do not contain an implementation of the good function. See Section 4.3 for more details on these test cases.)

For test cases that use multiple files, the following functions are defined in the “a” sub-file (e.g., CWE78_OS_Command_Injection__wchar_t_connect_socket_execl_51a.c). The “primary file” for a test case is a general term for the “a” sub-file in multi-file test cases, or the only file in single-file test cases.

4.1.1.1 Primary Bad Function

Each test case contains exactly one primary bad function in the primary file. In many simpler test cases, this function contains the flawed construct, but in other test cases this function calls other “sink” or “helper” function(s) that contain the flaw (“sink” and “helper” functions are described in a later section).

The primary bad function:

- For C, is named with the test case name followed by the string “_bad,” such as “CWE78_OS_Command_Injection__char_connect_socket_execl_01_bad(.”
- For C++, is named bad() and is in a namespace that is unique to the test case. The function is not part of a C++ class.
- Takes no parameters and has no return value.

The name of the primary bad function matches the following regular expression:

```
^(CWE.*_)?bad$
```

4.1.1.2 Primary Good Function

Each test case contains exactly one primary good function in the primary file (the same file as the primary bad function). The only code in this good function is a call to each of the secondary good functions (described in the next section). However, a few of the bad-only test cases contain empty good functions. This function does not contain any non-flawed constructs.

The primary good function:

- For C, this function is named with the test case name followed by the string “_good,” such as “CWE78_OS_Command_Injection__char_connect_socket_execl_01_good(.”
- For C++, this function is named good() and is in the namespace that is unique to the test case. The function is not part of a C++ class.
- Takes no parameters and has no return value.

The name of the primary good function matches the following regular expression:

```
^(CWE.*_)?good$
```

4.1.1.3 Secondary Good Function(s)

Non-class-based test cases also contain one or more secondary good functions in the primary file. Some of the bad-only test cases, however, do not include any secondary good functions. In many simpler test cases, these secondary good functions contain the actual non-flawed constructs. In other test cases, these functions will call “sink” or “helper” functions, which contain the non-flawed constructs. The number of secondary good functions depends on the test case’s flaw type as well as how many non-flawed constructs similar to that flaw exist. Many test cases have only one secondary good function, but others may have more.

There are three naming conventions used for secondary good functions:

- goodG2B, goodG2B1, goodG2B2, goodG2B3, etc. – These names are used in data flow test cases when a good source is passing safe data to a potentially bad sink.
- goodB2G, goodB2G1, goodB2G2, goodB2G3, etc. – These names are used in data flow test cases when a bad source is passing unsafe or potentially unsafe data to a good sink.
- good1, good2, good3, etc. – This is the “default” or “generic” name for these functions when the conditions above do not apply.

The names of the secondary good functions match the following regular expression:

```
^good(\d+|G2B\d*|B2G\d*)$
```

Note: It is important that this regular expression does not overlap with the previously defined good function regular expression so that the primary good functions are not matched.

The secondary good functions have the same argument and return types as the primary bad and primary good functions. In addition, the secondary good functions have the following characteristics:

- In C and C++ test cases, the secondary good functions are statically scoped. Therefore, they are only accessible within that source code file, which prevents name collisions.
- In C++ test cases, the secondary good functions are in the namespace that is unique to the test case. The functions are not part of a C++ class.

4.1.2 Optional Functions

In addition to the required functions, test cases may define “helper,” “source,” and/or “sink” functions as described in the following sections.

4.1.2.1 Helper Functions

Helper functions are used in test cases when even the simplest form of the flaw cannot be contained in a single function (within the constraints of the test case design). Functions used to create data flow patterns (“source” and “sink” functions) in More Complex test cases are not considered “helper” functions because they are not part of the flaw construct.

Examples of test cases where helper functions are required include:

- Test cases involving variable argument functions, such as in the C test case CWE134_Uncontrolled_Format_String__char_console_vprintf_01.
- Test cases for unused parameter, such as in the C test case CWE563_Unused_Variable__unused_parameter_variable_01.

The following items describe helper functions further:

- The helper functions are always specific to the bad function or good functions. The bad helper and good helper functions may contain different code or the exact same code (separate functions are used to easily evaluate tool results as “True Positives” or “False Positives”).
- Helper functions for the bad code are named “helperBad.”
- Ideally, helper functions would be specific to an individual secondary good function and be named like “helperGood1” or “helperGoodG2B.” This naming is used in manually created test cases, but unfortunately is not supported in the current Test Case Template Engine. In generated test cases, a generic function named “helperGood” is used.
- In C test cases, helper functions are statically scoped when possible.
- In C++ test cases, helper functions are in the namespace for the test case and are statically scoped when possible.
- In multi-file test cases, helper functions may be in the primary file or in the other, non-primary files.
- In test cases using variable argument functions, such as CWE134_Uncontrolled_Format_String__char_console_vprintf_01, helper functions are named “badVaSink,” “goodG2BVaSink,” and “goodB2GVaSink.” In control flow test cases, such as CWE134_Uncontrolled_Format_String__char_console_vprintf_02, helper functions are named “badVaSinkB,” “goodB2G1VaSinkG,” “goodB2G2VaSinkG,” “goodG2B1VaSinkB,” etc. Despite their names, these “VaSink” functions are considered “helper” functions rather than “sink” functions because they are needed in even the simplest forms of the flaws.

The names of the helper functions will match the following regular expressions:

```
^(CWE.+_)?(helperBad|badVaSink[BG]?)$
```

```
^(CWE.+_)?((helperGood(G2B|B2G)?\d*)|(good(G2B|B2G)?\d*VaSink[BG]?)?)$
```

4.1.2.2 Source and Sink Functions

Test cases that contain data flows use “source” and “sink” functions, which are called from each other or from the primary bad or good function. Each source or sink function is specific to either the bad function for the test case or for exactly one secondary good function.

The following items describe source and sink functions further:

- Bad source and sink functions are generally named “BadSource” and “BadSink.”

- Good source functions are generally named “goodG2BSource,” “goodG2B1Source,” “goodB2GSource,” “goodB2G2Source,” etc.
- Good sink functions are generally named “goodG2BSink,” “goodG2B1Sink,” “goodB2GSink,” “goodB2G2Sink,” etc.
- In C test cases, source and sink function are statically scoped when possible. If the function is called from another file, then the test case name is added to the beginning of the function name. Examples of non-static source and sink function names are “CWE134_Uncontrolled_Format_String__char_console_printf_61b_badSource” and “CWE134_Uncontrolled_Format_String__char_console_printf_51b_goodG2BSink.”
- In C++, source and sink functions are defined in the namespace for that test case and are statically scoped when possible. When multiple source or multiple sink functions are used in a test case, they are named “badSink_b,” badSink_c,” etc. and “goodG2BSink_b,” “goodG2BSink_c,” etc.
- In multi-file test cases, source and sink functions may be defined in the primary file or in the other, non-primary files.

The names of the source and sink functions will match the following regular expressions:

```
^(CWE.+_)badSource(_[a-z])?$
```

```
^(CWE.+_)badSink(_[a-z])?$
```

```
^(CWE.+_)good(G2B\d*|B2G\d*)?Source(_[a-z])?$
```

```
^(CWE.+_)good(G2B\d*|B2G\d*)?Sink(_[a-z])?$
```

4.2 Class-Based Flaw Test Cases

The design of test cases for C++ class-based flaws (i.e., those flaws that affect an entire class and not just a statement or code block) are slightly different because the bad and good constructs cannot be contained in an arbitrary function. These test cases use separate classes in separate files.

Bad File for Class-Based Flaws

In a test case for a class-based flaw, the bad file:

- Has a name that ends in `_bad` (before the extension). For example, “CWE401_Memory_Leak__destructor_01_bad.cpp.”
- Contains a required bad function with a signature like the bad function in a test case for a non-class-based flaw. This function makes use of the bad class for this test case to exercise the flaw being tested.

- In C++, this function is in the namespace for the test case, but outside of the class in the file.
- If there is only one class in the file, it is named “BadClass.” If a base class and a subclass are needed for the flaw, the classes are in the same file and are named “BadBaseClass” and “BadDerivedClass.”
- Has a main function that calls the bad function. Like the main functions in test cases for non-class-based flaws, this function is only used for testing or building separate binaries for the test case.

Good File for Class-Based Flaws

In a test case for a class-based flaw, the good file:

- Has a name that ends in “_good1” (before the extension). For example, “CWE401_Memory_Leak__destructor_01_good1.cpp.” Future versions of the test cases may include additional good files with names containing “_good2,” “_good3,” etc.
- Contains a required primary good function named “good” with a signature like the “good” function in a test case for a non-class-based flaw. Like the primary good function in a test case for a non-class-based flaw, this function only calls the secondary good function in this file.
 - In C++, this function is in the namespace for the test case, but outside of the class in the file.
- Contains at least one required secondary good function named “good1” to match the file name (currently, only “good1” function names are used, but future versions of the test cases may use functions “good2,” “good3,” etc.). The signature for this function is like the signature of a secondary good function in a test case for a non-class-based flaw. This secondary good function makes use of the class in this file to exercise the non-flawed construct being tested.
 - In C++, the “good1” function is statically scoped and in the namespace for the test case, but outside of the class in the file.
- In C++, if there is only one class in the file, it is named “GoodClass.” If a base class and a subclass are needed for the non-flawed construct, the classes are in the same file and are named “GoodBaseClass” and “GoodDerivedClass.”
- Has a main function that calls the primary good function. Like the main functions in test cases for non-class-based flaws, this function is only used for testing or building separate binaries for the test case.

4.3 Virtual Function Test Cases

A few test cases, like ones using flow variants 81 and 82, make use of virtual functions. In order to fit these types of test cases into the test case suite, they are designed slightly different than the “traditional” test cases described in the previous sections.

A C++ virtual function Data Flow test case contains five files:

1. Header file – This file defines the base class and declares a function, named “action,” within the base class as a pure virtual function. It also defines bad and good classes and declares “action” functions within those that will implement the “action” function from the base class. The extension for this file is the standard “.h.”
2. Root file – This file contains the implementations for the bad and good functions. The file name contains the letter ‘a’ as a sub-file identifier and is a C++ source file.
3. Bad implementation file – This file implements the “action” function for the bad class, contains the string “bad” as a sub-file identifier, and is a C++ source file.
4. GoodG2B implementation file - This file implements the “action” function for a good class that makes use of a bad sink. The root file ensures that a good source is used with this bad sink. The file name contains the string “goodG2B” as a sub-file identifier, and is a C++ source file.
5. GoodB2G implementation file - This file implements the “action” function for a good class that makes use of a good sink. The root file ensures that a bad source is used with this good sink. The file name contains the string “goodB2G” as a sub-file identifier, and is a C++ source file.

As an example, the files for the CWE191_Integer_Underflow__unsigned_int_rand_sub_81 test case are as follows:

- CWE191_Integer_Underflow__unsigned_int_rand_sub_81.h
- CWE191_Integer_Underflow__unsigned_int_rand_sub_81a.cpp
- CWE191_Integer_Underflow__unsigned_int_rand_sub_81_bad.cpp
- CWE191_Integer_Underflow__unsigned_int_rand_sub_81_goodG2B.cpp
- CWE191_Integer_Underflow__unsigned_int_rand_sub_81_goodB2G.cpp

4.4 Constructor/Destructor Test Cases

A few test cases, like ones using flow variants 83 and 84, contain data flows between the constructor and destructor of a class. The code containing the “source” of the data flow is contained within the constructor and the code containing the “sink” of the data flow is contained within the destructor. Like the virtual function test cases described in Section 4.3, these types of test cases are designed slightly different than the “traditional” test cases in order for them to fit into the test case suite.

A C++ constructor/destructor Data Flow test case contains five files:

1. Header file – This file defines the bad and good classes. Each of these classes contains a single constructor and destructor. The extension for this file is the standard “.h.”
2. Root file – This file contains the implementations for the bad and good functions. These functions create objects that will call the constructors/destructors implemented in the remaining implementation files described below. The file name contains the letter “a” as a sub-file identifier and is a C++ source file.
3. Bad implementation file – This file implements the constructor and destructor for the bad class, contains the string “bad” as a sub-file identifier, and is a C++ source file.
4. GoodG2B implementation file – This file implements the constructor and destructor for a good class that makes use of a bad sink. The good source implementation is contained within the constructor and the bad sink implementation is contained within the destructor. The file name contains the string “goodG2B” as a sub-file identifier, and is a C++ source file.
5. GoodB2G implementation file – This file implements the constructor and destructor for a good class that makes use of a good sink. The bad source implementation is contained within the constructor and the good sink implementation is contained within the destructor. The file name contains the string “goodB2G” as a sub-file identifier, and is a C++ source file.

4.5 Bad-only Test Cases

During the test case design process, it was determined that in a few cases a non-flawed construct could not be generated that correctly fixed the flaw being tested. Therefore, a minimal number of test cases are considered “bad-only” in the sense that they only contain a flawed construct.

The bad-only test cases differ from the rest of the test cases in the following ways:

- All bad-only test cases are non-class-based.
- No bad-only test cases contain Data Flows.

The bad-only test cases follow the same naming scheme as non-class-based test cases. It should be noted that these test cases should be excluded from any analysis that attempts to determine the number of False Positives reported by static analysis tools. A list of these test cases appears in Appendix D.

Section 5: Test Case Support Files

As mentioned in Section 3.4 above, test cases are not self-contained. Every test case requires at least one common test case support file. There are additional test case support files that are CWE entry specific and used by test cases, where appropriate. In addition, support files with an included main function are provided to execute the test cases.

The sections below describe the purpose and contents of each test case support file.

5.1 Common Support Files

One or more common support files are required for every test case and are located in the *~testcasesupport* directory.

The standard test case header:

- *std_testcase.h* – This header file is included in every C/C++ test case source code file and contains several variable and macro definitions. It also includes other header files such as “*std_testcase_io.h*” and the system header “*stdio.h*” so that they don’t need to be included by every test case.

Input/Output related support files:

- *io.c* – This file contains definitions for several functions used by the test case source code files to print various types to the console. For example, *printLine()* is used to print an array of characters to the console. Test cases use the functions in this file instead of calling console output functions directly in order to prevent incidental issue reports from analysis tools for “inappropriate logging” or “possible data disclosure.” This file also contains the definitions of several global variables used by the Control Flow test cases.
- *std_testcase_io.h* – This header file declares the functions and variables which are defined in *io.c*. It is not named *io.h* because there is a system header in Windows with that name.

Thread related support files:

- *std_thread.c* – This file contains implementations of several thread-related functions used by the test case source code files.
- *std_thread.h* – This header file is used to define the functions in *std_thread.c*.

5.2 CWE Entry Specific Support Files

In addition to the common support files, test cases may make use of support files that are specific to multiple test cases associated with a CWE entry. When present, these files will be in the

directory for the CWE entry and will have a name that does not match the expected pattern for a test case file.

5.3 Main Support Files

Support files are also provided to test an individual CWE entry's test cases. These files, called `main.cpp` and `testcases.h`, are auto-generated and are included with each CWE entry (such as in the `~testcases\CWE15_External_Control_of_System_or_Configuration_Setting` directory). They can be used to test all the test cases contained within that CWE entry's directory.

Beginning with v1.2 of the Juliet Test Suite for C/C++, several CWE entries were split among multiple subdirectories due to the vast number of files. Each subdirectory is limited to a maximum of 1,000 test case files and contains a `main.cpp` file and a `testcases.h` file. These files can be used to compile and test all of the test cases contained within that subdirectory.

One "master" version of each file, included in the `~testcasesupport` directory, is used to run all the test cases at once.

Each file is described below:

- *main.cpp* – This file is auto-generated and contains a "main" function that calls the primary "good" function for each test case, and then calls the primary "bad" function for each test case. This file can be compiled using the preprocessor macro `OMITBAD` or `OMITGOOD` to omit the calls to the bad or good functions.
- *testcases.h* – This header file is automatically generated and contains declarations of the bad and good functions in the test cases so that they can be called by the `main.cpp` file. This header is only included in `main.cpp`.

Most of the test cases should compile on operating systems other than Windows. Test cases that are Windows-specific contain the string "w32" in their file names. The following support files are included in the directory (or directories) for each CWE entry if the directory contains test cases that will compile on Linux³:

- *main_linux.cpp* – This file is auto-generated and contains a "main" function that calls the primary "good" function, and then the primary "bad" function, for each test case that is not Windows-specific. This file can be compiled using the preprocessor macro `OMITBAD` or `OMITGOOD` to omit the calls to the "bad" or "good" functions.
- *Makefile* – A standard makefile that will compile all non-Windows-specific test cases within a CWE directory when executed by the utility, "make."

³ The test cases are not guaranteed to compile on Linux and were designed specifically for Windows. All test cases have been tested and compile successfully on Windows.

See Section 7.1 for details on how to update `main.cpp`, `main_linux.cpp`, `testcases.h`, and `Makefile` using scripts distributed with the test cases.

Section 6: Building Test Cases

6.1 Build Prerequisites

All files needed to build the test cases are included in this distribution using the following environment (development and test was done using versions shown in parentheses):

- Microsoft Windows platform (Windows 7)
- Microsoft Visual Studio (2010 Professional)
- Python for Windows (version 3.2.3)

The current release of the test cases targets the Microsoft Windows platform; however, many test cases will work on non-Windows platforms. Windows-specific test case files contain the string “w32” in their name.

Although the versions listed above were used to develop and test the test cases, other versions may work as well.

6.2 Compiling Test Cases

There are two ways to compile these test cases: as a single compiled executable that contains all test cases; or as individual executables, each containing a single CWE entry.

Note that some CWE entries’ test cases are split between multiple subdirectories. Compiling the code using the main.cpp and testcases.h in each of these subdirectories will only compile the test cases in that folder and not all of the test cases for the CWE entry. Also, due to the number of files and the number of lines of code contained in the test cases, some static analysis tools may not be able to analyze the single compiled executable.

6.2.1 Compile All Test Cases as One Executable

To compile a single (large) executable file named “testcases.exe,” run the file “compile_all.bat” located in the top level directory. This batch file must be run with Visual Studio specific environment variables set, which is most easily done by running it in a “Visual Studio Command Prompt.” This batch file can be used as a basis for analyzing all test cases by following instructions in the documentation for the tool being used.

The non-Windows-specific test cases can be compiled into a single (large) executable file named “all-testcases” by running “make” and targeting “Makefile_all,” located in the top level directory.

6.2.2 Compile Test Cases per CWE Entry

The test cases can also be compiled so that a separate executable file is generated for each CWE entry, with a few exceptions. This is accomplished by running the batch file in the directory for that CWE entry (such as by running “CWE476.bat” in the *~testcases\CWE476_NULL_Pointer_Dereference* directory to create the file “CWE476.exe”).

In order to automate the process of compiling the individual test cases in each CWE entry’s directory, the Python script “run_analysis_example_tool.py” can be executed (also in a “Visual Studio Command Prompt”). This script will go to each CWE entry directory and run the batch file to compile those test cases. This script can also be used as the basis for a script to automate performing analysis on the test cases for each CWE entry. The comments in the script provide an example of how this can be accomplished.

The test cases for a given CWE can also be compiled by running “make” within the CWE directory. Note, however, that there are several CWE directories that do not contain make files as all of the test cases for those CWEs are Windows-specific.

6.2.2.1 CWE Entries Containing Subdirectories

Due to the vast number of test case files for some CWE entries, test case files for these CWEs are split into subdirectories containing no more than 1,000 test case files per directory. For example, the test cases for CWE 590 are broken up into the following subdirectories:

- *~testcases\CWE590_Free_Memory_Not_on_Heap\s01*
- *~testcases\CWE590_Free_Memory_Not_on_Heap\s02*
- *~testcases\CWE590_Free_Memory_Not_on_Heap\s03*
- *~testcases\CWE590_Free_Memory_Not_on_Heap\s04*
- *~testcases\CWE590_Free_Memory_Not_on_Heap\s05*

Each subdirectory contains a batch file that can be used to compile all of the test case files located within that directory (for example, “CWE590_s01.bat”). The batch file, and generated executable, each contain the CWE number and subdirectory number in their file names. For example, “CWE590_s01.bat” can be executed to compile the test cases in the *~testcases\CWE590_Free_Memory_Not_on_Heap\s01* directory into “CWE590_s01.exe.”

Note that all flow variants for a given functional variant will appear within the same subdirectory.

6.3 Compiling an Individual Test Case

Although the test cases are typically compiled and analyzed in sets, the test cases are designed so that each test case can be compiled and executed individually. Running a test case is useful during test case development, but can also be used to analyze a test case in isolation.

6.3.1 Building and Running a Test Case

In the test cases, a main function exists that contains a call to the primary good function for the test case, followed by a call to the bad function. The INCLUDEMAIN preprocessor macro definition is used to include the main function when compiling a test case. The preprocessor macros OMITBAD and OMITGOOD can also be used to omit the flawed or non-flawed constructs from the compilation. Omitting portions of the test case(s) allows for compiling a binary that contain only flaws or only non-flaws, which may be useful when testing binary analysis tools.

The following example command will use the Visual Studio command line compiler to compile a single file test case into the file testcase.exe. This command should be run in a “Visual Studio Command Prompt” in the directory to which the test case .zip file was extracted.

```
cl /Itestcasesupport /DINCLUDEMAIN /Fetestcase.exe testcasesupport\io.c
testcasesupport\std_thread.c
testcases\CWE78_OS_Command_Injection\s02\CWE78_OS_Command_Injection__ch
ar_console_system_01.c
```

The following example command will compile a multiple file test case into the file testcase.exe.

```
cl /Itestcasesupport /DINCLUDEMAIN /Fetestcase.exe testcasesupport\io.c
testcasesupport\std_thread.c
testcases\CWE78_OS_Command_Injection\s02\CWE78_OS_Command_Injection__ch
ar_console_system_54*.c
```

In both cases, this will produce an executable testcase.exe.

Section 7: Updating Build Files

Included in the test case distribution are scripts that can be used to update the test case build files if changes are made to the set of test cases. Using the test cases as distributed or after edits are made to existing test case files do not require the use of these scripts. These scripts are only needed if test case files are deleted from the set or new test cases are added. If new test cases are added to the test case set, care should be taken to follow the test case design in order to prevent errors in these scripts, in compilation, or in tool result analysis.

7.1 Updating C/C++ Build Files

The C/C++ test case archive contains three scripts that can be used to update the build files if changes are made to the set of test cases to be analyzed.

- *create_single_batch_file.py* – Running this script will update the file “compile_all.bat,” which can be run to compile all the test cases into a single executable file. This script also edits source code and header files needed for a successful compilation with this batch file.
- *create_single_Makefile.py* – Running this script will update the file “Makefile_all,” which can be targeted with “make” to compile all the non-Windows-specific test cases into a single executable file. This script also edits source code and header files needed for a successful compilation with this makefile.
- *create_per_cwe_files.py* – Running this script will update the batch files, and makefiles for non-Windows-specific test cases if they exist, that compile the test cases for each CWE entry into a separate executable file. This script also edits source code and header files needed for a successful compilation with those batch files.

Section 8: Tool Analysis

The test cases have been designed so that static analysis tool results can be easily evaluated. This section describes the desired results when running a static analysis tool on the test cases.

8.1 True Positives and False Negatives

When a static analysis tool is run on a test case, one desired result is for the tool to report one flaw of the target type. That reported flaw should be in a function with the word “bad” in its name (such as `bad()`, `badSink()`, or `CWE476_NULL_Pointer_Dereference__char_41_bad()`) or in a class whose implementation is contained within a file with the word “bad” in the file name (such as “`CWE401_Memory_Leak__destructor_01_bad.cpp`”). A correct report of this type is considered a “True Positive.”

In some circumstances, tools may report the flaw in a test case more than once in the bad functions or classes. For example, a tool may report multiple, slightly different flaw types or, in other cases, a tool may report flaws in different locations. At times, a tool may report two results with the exact same type in the exact same location (sometimes with different call stacks or other different metadata).

If the tool does not report a flaw of the target type in a bad function or class in a test case, then the result for the tool on the test case is considered a “False Negative.”

8.2 False Positives and True Negatives

The other desired result when running a tool on a test case is for the tool to not report any flaws of the target type in a function with the word “good” in its name or a class with the word “good” in its file name. An incorrect report of the target flaw type in a good function is considered a “False Positive.”

As described in Section 4.1.1.3, each non-class-based test case has one or more secondary good functions that contain a non-flawed construct. When a test case has more than one secondary function, a test case user may want to determine in which secondary good function(s) a tool reported False Positives and in which secondary good function(s) the tool had no False Positives (that is, where the tool had “True Negative(s)”).

In many test cases, this can be determined by examining the name of the functions where tool results are reported. The “source” and “sink” functions can be associated with the secondary good function from which they are called (for example, the function `goodB2GSource` or `goodB2GSink` can be associated with the secondary good function `goodB2G`).

Unfortunately, limitations of the CAS’s Test Case Template Engine used to generate test cases prevent being able to definitively tie all False Positive results to the secondary good functions in all test cases. Specifically, as detailed in Section 4.1.2.1, good helper functions are not specific to the secondary good functions in a test case. Therefore, in a test case with more than one

secondary good function and one or more False Positive results in a good helper function, the False Positive(s) cannot be easily associated with secondary good function(s) and True Negatives cannot be determined, either.⁴

8.3 Unrelated Flaw Reports

A tool may also report flaws with types that are not related to the target flaw type in a test case. There are two occasions when this may occur:

- Those flaw reports may be correctly pointing out flaws of the non-target type that are present in the test case. Flaws of this type are known as “incidental” flaws. The developers of the test cases attempted to minimize the incidental flaws and marked unavoidable incidental flaws with a comment containing the string “INCIDENTAL.” However, many uncommented incidental flaws remain in the test cases so users should not draw any conclusions about tool reports of non-target flaw types without investigating the reported result fully.
- The flaw reports may be indicating flaws that do not exist in the test case. Flaw reports of this type are known as “unrelated False Positives” because they are incorrect flaw reports (False Positives) and not related to the type of flaw the test case is intended to test.

Flaw reports of non-target types generally cannot be characterized as correct or incorrect in an automated or trivial manner. They may be triggered by common code constructs that are repeated in a large number of test cases (due to the automated generation process used to create the test cases). For these reasons, these flaw reports are typically ignored when studying a static analysis tool.

⁴ This association cannot be made based solely on function names. Some tools may report additional information, such as stack traces, with findings that allow this association to be made.

Appendix A: Test Case CWE Entries

The table below shows the CWE entries associated with the 2012 Test Cases, along with the number of test cases associated with each CWE entry.

CWE Entry ID	CWE Entry Name	C/C++ Test Cases
15	External Control of System or Configuration Setting	48
23	Relative Path Traversal	2400
36	Absolute Path Traversal	2400
78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	4800
90	Improper Neutralization of Special Elements used in an LDAP Query ('LDAP Injection')	480
114	Process Control	576
121	Stack-based Buffer Overflow	4968
122	Heap-based Buffer Overflow	5922
123	Write-what-where Condition	144
124	Buffer Underwrite ('Buffer Underflow')	2048
126	Buffer Over-read	1452
127	Buffer Under-read	2048
134	Uncontrolled Format String	2880
176	Improper Handling of Unicode Encoding	48
188	Reliance on Data/Memory Layout	36
190	Integer Overflow or Wraparound	2592
191	Integer Underflow (Wrap or Wraparound)	1584
194	Unexpected Sign Extension	1152
195	Signed to Unsigned Conversion Error	1152
196	Unsigned to Signed Conversion Error	18
197	Numeric Truncation Error	864
222	Truncation of Security-relevant Information	18
223	Omission of Security-relevant Information	18
226	Sensitive Information Uncleared Before Release	72
242	Use of Inherently Dangerous Function	18
244	Improper Clearing of Heap Memory Before Release ('Heap Inspection')	72
247	Reliance on DNS Lookups in a Security Decision	18
252	Unchecked Return Value	630
253	Incorrect Check of Function Return Value	684
256	Plaintext Storage of a Password	96
259	Use of Hard-coded Password	96
272	Least Privilege Violation	252
273	Improper Check for Dropped Privileges	36
284	Improper Access Control	216
319	Cleartext Transmission of Sensitive Information	192
321	Use of Hard-coded Cryptographic Key	96
325	Missing Required Cryptographic Step	72
327	Use of a Broken or Risky Cryptographic Algorithm	54
328	Reversible One-Way Hash	54
338	Use of Cryptographically Weak PRNG	18

364	Signal Handler Race Condition	18
366	Race Condition within a Thread	36
367	Time-of-check Time-of-use (TOCTOU) Race Condition	36
369	Divide By Zero	864
377	Insecure Temporary File	144
390	Detection of Error Condition Without Action	90
391	Unchecked Error Condition	54
396	Declaration of Catch for Generic Exception	54
397	Declaration of Throws for Generic Exception	20
398	Indicator of Poor Code Quality	181
400	Uncontrolled Resource Consumption ('Resource Exhaustion')	720
401	Improper Release of Memory Before Removing Last Reference ('Memory Leak')	1658
404	Improper Resource Shutdown or Release	384
415	Double Free	962
416	Use After Free	459
426	Untrusted Search Path	192
427	Uncontrolled Search Path Element	480
440	Expected Behavior Violation	1
457	Use of Uninitialized Variable	948
459	Incomplete Cleanup	36
464	Addition of Data Structure Sentinel	48
467	Use of sizeof() on a Pointer Type	54
468	Incorrect Pointer Scaling	37
469	Use of Pointer Subtraction to Determine Size	36
475	Undefined Behavior For Input to API	36
476	NULL Pointer Dereference	348
478	Missing Default Case in Switch Statement	18
479	Signal Handler Use of a Non-reentrant Function	18
480	Use of Incorrect Operator	18
481	Assigning instead of Comparing	18
482	Comparing instead of Assigning	18
483	Incorrect Block Delimitation	20
484	Omitted Break Statement in Switch	18
500	Public Static Field Not Marked Final	1
506	Embedded Malicious Code	158
510	Trapdoor	70
511	Logic/Time Bomb	72
526	Information Exposure Through Environmental Variables	18
534	Information Exposure Through Debug Log Files	36
535	Information Exposure Through Shell Error Message	36
546	Suspicious Comment	90
561	Dead Code	2
562	Return of Stack Variable Address	3
563	Unused Variable	512
570	Expression is Always False	16
571	Expression is Always True	16
587	Assignment of a Fixed Address to a Pointer	18
588	Attempt to Access Child of a Non-structure Pointer	80
590	Free of Memory not on the Heap	2680
591	Sensitive Data Storage in Improperly Locked Memory	96

605	Multiple Binds to Same Port	18
606	Unchecked Input for Loop Condition	480
615	Information Exposure Through Comments	18
617	Reachable Assertion	306
620	Unverified Password Change	18
665	Improper Initialization	193
666	Operation on Resource in Wrong Phase of Lifetime	90
667	Improper Locking	18
672	Operation on a Resource after Expiration or Release	47
674	Uncontrolled Recursion	2
675	Duplicate Operations on Resource	192
676	Use of Potentially Dangerous Function	18
680	Integer Overflow to Buffer Overflow	576
681	Incorrect Conversion between Numeric Types	54
685	Function Call With Incorrect Number of Arguments	18
688	Function Call With Incorrect Variable or Reference as Argument	18
690	Unchecked Return Value to NULL Pointer Dereference	960
758	Reliance on Undefined, Unspecified, or Implementation-Defined Behavior	581
761	Free of Pointer not at Start of Buffer	576
762	Mismatched Memory Management Routines	3564
773	Missing Reference to Active File Descriptor or Handle	144
775	Missing Release of File Descriptor or Handle after Effective Lifetime	144
780	Use of RSA Algorithm without OAEP	18
785	Use of Path Manipulation Function without Maximum-sized Buffer	18
789	Uncontrolled Memory Allocation	960
832	Unlock of a Resource that is not Locked	18
835	Loop with Unreachable Exit Condition ('Infinite Loop')	6
843	Access of Resource Using Incompatible Type ('Type Confusion')	80

Table 3 – CWE Entries in 2012 Test Cases

Appendix B: CWE/SANS Top 25 Coverage

Table 4 shows the degree to which the C/C++ test cases cover each of the 2011 CWE/SANS Top 25 Most Dangerous Software Errors.

Note: As of this writing, 2011 is the most recent version.

2011 CWE/SANS Top 25		CAS Test Cases	
Rank	CWE Entry	CWE Entry / Entries	C/C++
1	CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	None (SQL Injection issues are covered in the related Java test cases)	-
2	CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	CWE-78	4800
3	CWE-120: Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')	CWE-121: Stack-based Buffer Overflow, CWE-122: Heap-based Buffer Overflow, CWE-680: Integer Overflow to Buffer Overflow	11466
4	CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	None (Cross-Site Scripting issues are covered in the related Java test cases)	-
5	CWE-306: Missing Authentication for Critical Function	None (Design issue which does not fit into CAS Test Case structure)	-
6	CWE-862: Missing Authorization	None (Design issue which does not fit into CAS Test Case structure)	-
7	CWE-798: Use of Hard-coded Credentials	CWE-259: Use of Hard-coded Password, CWE-321: Use of Hard-coded Cryptographic Key	192
8	CWE-311: Missing Encryption of Sensitive Data	CWE-319: Cleartext Transmission of Sensitive Information	192
9	CWE-434: Unrestricted Upload of File with Dangerous Type	None (Design issue which does not fit into CAS Test Case structure)	-
10	CWE-807: Reliance on Untrusted Inputs in a Security Decision	CWE-247: Reliance on DNS Lookups in a Security Decision	18
11	CWE-250: Execution with Unnecessary Privileges	None (Design issue which does not fit into CAS Test Case structure)	-

2011 CWE/SANS Top 25		CAS Test Cases	
Rank	CWE Entry	CWE Entry / Entries	C/C++
12	CWE-352: Cross-Site Request Forgery (CSRF)	None (Design issue which does not fit into CAS Test Case structure)	-
13	CWE-22: Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	CWE-23: Relative Path Traversal, CWE-36: Absolute Path Traversal	4800
14	CWE-494: Download of Code Without Integrity Check	None (Design issue which does not fit into CAS Test Case structure)	-
15	CWE-863: Incorrect Authorization	None (Design issue which does not fit into CAS Test Case structure)	-
16	CWE-829: Inclusion of Functionality from Untrusted Control Sphere	None (Design issue which does not fit into CAS Test Case structure)	-
17	CWE-732: Incorrect Permission Assignment for Critical Resource	None (Design issue which does not fit into CAS Test Case structure)	-
18	CWE-676: Use of Potentially Dangerous Function	CWE-676	18
19	CWE-327: Use of a Broken or Risky Cryptographic Algorithm	CWE-327	54
20	CWE-131: Incorrect Calculation of Buffer Size	CWE-121: Stack-based Buffer Overflow, CWE-122: Heap-based Buffer Overflow	288
21	CWE-307: Improper Restriction of Excessive Authentication Attempts	None (Design issue which does not fit into CAS Test Case structure)	-
22	CWE-601: URL Redirection to Untrusted Site ('Open Redirect')	None (Open Redirect issues are covered in the related Java test cases)	-
23	CWE-134: Uncontrolled Format String	CWE-134	2880
24	CWE-190: Integer Overflow or Wraparound	CWE-190, CWE-191: Integer Underflow (Wrap or Wraparound)	4176
25	CWE-759: Use of a One-Way Hash without a Salt	None (Use of a One-Way Hash without a Salt issues are covered in the related Java test cases)	-

Table 4 – Test Case Coverage of 2011 CWE/SANS Top 25

Appendix C: Test Case Flow Variants

Below is a table containing information about the Flow Variants in the C/C++ test cases, including a brief description. Due to design constraints, all flow types do not contain test cases for each flow variant.

Flow Variant	Flow Type	Description	C	C++
01	None	Baseline – Simplest form of the flaw	X	X
02	Control	if(1) and if(0)	X	X
03	Control	if(5==5) and if(5!=5)	X	X
04	Control	if(STATIC_CONST_TRUE) and if(STATIC_CONST_FALSE)	X	X
05	Control	if(staticTrue) and if(staticFalse)	X	X
06	Control	if(STATIC_CONST_FIVE==5) and if(STATIC_CONST_FIVE!=5)	X	X
07	Control	if(staticFive==5) and if(staticFive!=5)	X	X
08	Control	if(staticReturnsTrue()) and if(staticReturnsFalse())	X	X
09	Control	if(GLOBAL_CONST_TRUE) and if(GLOBAL_CONST_FALSE)	X	X
10	Control	if(globalTrue) and if(globalFalse)	X	X
11	Control	if(globalReturnsTrue()) and if(globalReturnsFalse())	X	X
12	Control	if(globalReturnsTrueOrFalse())	X	X
13	Control	if(GLOBAL_CONST_FIVE==5) and if(GLOBAL_CONST_FIVE!=5)	X	X
14	Control	if(globalFive==5) and if(globalFive!=5)	X	X
15	Control	switch(6) and switch(7)	X	X
16	Control	while(1)	X	X
17	Control	for loops	X	X
18	Control	goto statements	X	X
21	Control	Flow controlled by value of a static global variable. All functions contained in one file.	X	X
22	Control	Flow controlled by value of a global variable. Sink functions are in a separate file from sources.	X	X
31	Data	Data flow using a copy of data within the same function	X	X
32	Data	Data flow using two pointers to the same value within the same function	X	X
33	Data	Use of a C++ reference to data within the same function	*	X
34	Data	Use of a union containing two methods of accessing the same data (within the same function)	X	X
41	Data	Data passed as an argument from one function to another in the same source file	X	X
42	Data	Data returned from one function to another in the same source file	X	X
43	Data	Data flows using a C++ reference from one function to another in the same source file	*	X
44	Control/ Data	Data passed as an argument from one function to a function in the same source file called via a function pointer	X	X
45	Data	Data passed as a static global variable from one function to another in the same source file	X	X
51	Data	Data passed as an argument from one function to another in different source files	X	X
52	Data	Data passed as an argument from one function to another to another in three different source files	X	X

Flow Variant	Flow Type	Description	C	C++
53	Data	Data passed as an argument from one function through two others to a fourth; all four functions are in different source files	X	X
54	Data	Data passed as an argument from one function through three others to a fifth; all five functions are in different source files	X	X
61	Data	Data returned from one function to another in different source files	X	X
62	Data	Data flows using a C++ reference from one function to another in different source files	*	X
63	Data	Pointer to data passed from one function to another in different source files	X	X
64	Data	void pointer to data passed from one function to another in different source files	X	X
65	Control/ Data	Data passed as an argument from one function to a function in a different source file called via a function pointer	X	X
66	Data	Data passed in an array from one function to another in different source files	X	X
67	Data	Data passed in a struct from one function to another in different source files	X	X
68	Data	Data passed as a global variable in the "a" class from one function to another in different source files	X	X
72	Data	Data passed in a vector from one function to another in different source files	*	X
73	Data	Data passed in a linked list from one function to another in different source files	*	X
74	Data	Data passed in a hash map from one function to another in different source files	*	X
81	Data	Data passed in an argument to a virtual function called via a reference	*	X
82	Data	Data passed in an argument to a virtual function called via a pointer	*	X
83	Data	Data passed to a class constructor and destructor by declaring the class object on the stack	*	X
84	Data	Data passed to a class constructor and destructor by declaring the class object on the heap and deleting it after use	*	X

* Included in C test cases as C++ files (logic requires C++ features)

Table 5 – Test Case Flow Variants

Appendix D : Bad-Only Test Cases

CWE Entry ID	CWE Entry Name	Functional Variants	Flow Variants
506	Embedded Malicious Code	email file_transfer_connect_socket file_transfer_listen_socket screen_capture	*
510	Trapdoor	network_connection network_listen	*

* Applicable to all variants.

Table 6 – C/C++ Bad-only Test Cases

Appendix E: Test Case Changes in v1.2

Test cases for the following CWEs were added in Juliet Test Suite v1.2 for C/C++:

CWE Entry ID	CWE Entry Name
90	Improper Neutralization of Special Elements used in an LDAP Query ('LDAP Injection')
176	Improper Handling of Unicode Encoding
526	Information Exposure Through Environmental Variables
615	Information Exposure Through Comments
667	Improper Locking
681	Incorrect Conversion between Numeric Types

Table 7 – CWEs Added in Juliet Test Suite v1.2 for C/C++

During quality control review, the test cases for the following CWEs were determined to be invalid and were removed in Juliet Test Suite v1.2 for C/C++:

CWE Entry ID	CWE Entry Name
204	Response Discrepancy Information Exposure
304	Missing Critical Step in Authentication
374	Passing Mutable Objects to an Untrusted Method
392	Missing Report of Error Condition
489	Leftover Debug Code
547	Use of Hard-coded, Security-relevant Constants
560	Use of umask() with chmod-style Argument

Table 8 – CWEs Removed in Juliet Test Suite v1.2 for C/C++

The following flow variants were added in Juliet Test Suite v1.2 for C/C++:

Flow Variant Number	Flow Type	Description
83	Data	Data passed to a class constructor and destructor by declaring the class object on the stack
84	Data	Data passed to a class constructor and destructor by declaring the class object on the heap and deleting it after use

Table 9 – Flow Variants Added in Juliet Test Suite v1.2 for C/C++

The following flow variant was removed in Juliet Test Suite v1.2 for C/C++:

Flow Variant Number	Flow Type	Description	Reason for Removal
19	Control	Dead code after a return	Reduce incidental dead code in the test suite

Table 10 – Flow Variants Removed in Juliet Test Suite v1.2 for C/C++

The number of flaw types for the following CWEs changed in Juliet Test Suite v1.2 for C/C++. Reasons for these changes include, but are not limited to moving the flaw type to a more specific CWE, removing the flaw type entirely, or adding additional flaw types.

CWE Entry ID	CWE Entry Name	Flaws in v1.1	Flaws in v1.2	Increase/Decrease
23	Relative Path Traversal	30	50	+20
36	Absolute Path Traversal	30	50	+20
122	Heap-based Buffer Overflow	123	126	+3
244	Improper Clearing of Heap Memory Before Release ('Heap Inspection')	2	4	+2
272	Least Privilege Violation	10	14	+4
284	Improper Access Control	10	12	+2
328	Reversible One-Way Hash	2	3	+1
398	Indicator of Poor Code Quality	6	11	+5
400	Uncontrolled Resource Consumption ('Resource Exhaustion')	10	15	+5
415	Double Free	19	22	+3
416	Use After Free	19	22	+3
457	Use of Uninitialized Variable	44	45	+1
476	NULL Pointer Dereference	6	10	+4
563	Unused Variable	23	26	+3
570	Expression is Always False	6	16	+10
571	Expression is Always True	6	16	+10
590	Free of Memory not on the Heap	57	67	+10
690	Unchecked Return Value to NULL Pointer Dereference	17	20	+3
758	Reliance on Undefined, Unspecified, or Implementation-Defined Behavior	33	37	+4
762	Mismatched Memory Management Routines	76	86	+10
773	Missing Reference to Active File Descriptor or Handle	6	3	-3
775	Missing Release of File Descriptor or Handle after Effective Lifetime	4	3	-1

Table 11 – Flaw Count Changes in Juliet Test Suite v1.2 for C/C++