Juliet Test Suite v1.2 for Java

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 Document Purpose

This document describes the Juliet Test Suite v1.2 for Java. 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 instructions 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 Java test case "CWE476_NULL_Pointer_Dereference__int_array_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 4.4).

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 (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 method 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 methods available on the underlying platform rather than the use of third-party libraries.

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).

2.2 Third Party Libraries

The test cases limit the use of features to those found in Java 1.6. Currently, the test cases cover issues that can affect standalone Java applications or Java Servlets. No test cases specifically cover Java Applets or Java Server Pages (JSPs) at this time.

The Java Servlet test cases make use of the Java Servlet API version 2.5 or above. The test cases were developed and are distributed with Apache's implementation of this API in the file servlet-api.jar. However, the test cases should build and run with any implementation of version 2.4 or greater of the Servlet API.

Additional third party libraries were used when necessary, but their usage was limited. API calls in these libraries were only used if there was no other conceivable way to implement a test case for a given flaw type. Table 1 lists these libraries and the names of the files included in the test case suite.

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

Library	File
Apache Commons Codec 1.5 API	commons-code-1.5.jar
Apache Commons Lang 2.5 API	commons-lang-2.5.jar
JavaMail API 1.4.4	javamail-1.4.4.jar

Table 1 – Additional Java Libraries

2.3 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, two are issues specific to the Buffer Handling weakness class, which is not relevant to Java, and two are covered in the related C/C++ 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 Java. The number of Java test cases in 2012 totaled 25,477, as opposed to 23,957 in 2011. This represents an increase of over 6%. Table 2 contains statistics on the size and scope of the test cases for 2011 and 2012.

	2011	2012	Percentage Increase
CWE Entries Covered	113	112	-0.90%
Flaw Types	751	841	12.00%
Test Cases	23,957	25,477	6.30%
Lines of Code[2]	4,712,718	4,565,713	-3.12%

Table 2 – 2011-2012 Java 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 methods.

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

- Test cases for an additional seven CWEs were added.
- Test cases for eight CWEs were removed.
- One flow variant was removed.
- The number of flaw types for twenty 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 name (sometimes in a shortened form) 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 for the test case files (".java").

The name for a test case is written as "Java test case CWE563_Unused_Variable__unused_ init_variable_int_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 method 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:

- "Servlet" – This string in the functional variant name for a test case indicates that the test case is a Java Servlet and inherits AbstractTestCaseServlet. An example of such a test case is the Java test case CWE78_OS_Command_Injection__getParameter_Servlet_01.

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 Java 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 could 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 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:

Java test case CWE476_NULL_Pointer_Dereference__int_array_01 consists of one file:

- CWE476_NULL_Pointer_Dereference__int_array_01.java

Java test case CWE476_NULL_Pointer_Dereference__int_array_22 consists of two files:

- CWE476_NULL_Pointer_Dereference__int_array_22a.java
- CWE476_NULL_Pointer_Dereference__int_array_22b.java

Java test case CWE476_NULL_Pointer_Dereference__Integer_54 consists of five files:

- CWE476_NULL_Pointer_Dereference__Integer_54a.java
- CWE476_NULL_Pointer_Dereference__Integer_54b.java
- CWE476_NULL_Pointer_Dereference__Integer_54c.java
- CWE476_NULL_Pointer_Dereference__Integer_54d.java
- CWE476_NULL_Pointer_Dereference__Integer_54e.java

Java test case CWE563_Unused_Variable__unused_public_member_variable_01 consists of two files:

- CWE563_Unused_Variable__unused_public_member_variable_01_bad.java
- CWE563_Unused_Variable__unused_public_member_variable_01_good1.java

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 "java"	Mandatory

Table 3 – 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 "readLine" method 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: "byte_console_readLine_add"
Flow Variant: 01
Language: Java

The test case will be contained in the file named:

CWE190_Integer_Overflow__byte_console_readLine_add_01.java

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 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 "CWE665_Improper_Initialization__flag_01_bad.java") and the non-flaw will be in the file identified with the string "_good1" (such as "CWE665_Improper_Initialization__flag_01_good1"). Section 4.2 contains more information about class-based flaws.

- Some Data Flow test cases involve the flow of data between methods in different classes. In these test cases, the test case will "start" in the class identified with the string "a," such as in the file "CWE90_LDAP_Injection__connect_tcp_54a.java." Methods in the "a" class will call methods in the "b" class, which may call methods in the "c" class, etc.

- Some Data Flow test cases involve the flow of data between abstract method calls. A base class is used to define the abstract method and implementations occur in separate classes.

Section 4: Test Case Design

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

> Java test case CWE581_Object_Model_Violation__hashCode_01
>
> (In this test case, failure to override the hashCode() method when overriding the equals() method could lead to incorrect equality calculations.)

Abstract method and bad-only test cases are unique. Abstract method 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 test cases also define a "main" method in the primary file. This main method 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 archives to use in testing binary analysis tools.

The sections below describe the test case design for non-class-based flaw, class-based flaw, abstract method, and bad-only test cases.

4.1 Non-Class-Based Flaw Test Cases

4.1.1 Required Methods

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

For test cases that use multiple files, the following methods are defined in the "a" sub-file (e.g., CWE78_OS_Command_Injection__connect_tcp_51a.java). 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.

The class for the test case (or the class in the "a" sub-file for test cases with multiple files) inherits AbstractTestCase or AbstractTestCaseServlet, which will cause the Java compiler to enforce the presence and signature of the primary methods below. The class which implements AbstractTestCase or AbstractTestCaseServlet is known as the "primary class" for the test case.

4.1.1.1 Primary Bad Method

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

The primary bad method:

- Is always named bad().
- Is a public method.
- For non-Servlet test cases, takes no arguments and has no return value.
- For Servlet test cases, has no return value and takes two arguments: an HttpServletRequest and an HttpServletResponse. Those arguments allow the test case to access the HTTP request received from the client and to write to the HTTP response which is sent to the client.

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

^bad$

4.1.1.2 Primary Good Method

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

The primary good method:

- Is always named good().
- Is a public method.
- For non-Servlet test cases, this method takes no arguments and has no return value.
- For Servlet test cases, this method has no return value and takes two arguments: an HttpServletRequest and an HttpServletResponse. Those arguments allow the test case to access the HTTP request received from the client and to write to the HTTP response which is sent to the client.

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

^good$

4.1.1.3 Secondary Good Method(s)

Non-class-based test cases also contain one or more secondary good methods in the primary file. Some of the bad-only test cases, however, do not include any secondary good methods. In many simpler test cases, these secondary good methods contain the actual non-flawed constructs. In other test cases, these methods will call "sink" or "helper" methods, which contain the non-flawed constructs. The number of secondary good methods 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 method, but others may have more.

There are three naming conventions used for secondary good methods:

- 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 methods when the conditions above do not apply.

The names of the secondary good methods 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 method's regular expression so that the primary good methods are not matched.

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

- The secondary good methods are private methods in the class for the test case.

4.1.2 Optional Methods

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

4.1.2.1 Helper Methods

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

An example of a test case where helper methods are required:

- Test cases for unused parameter, such as in the Java test case CWE563_Unused_Variable__unused_parameter_value_01.

The following items describe helper methods further:

- The helper methods are always specific to the bad or good methods. The bad helper and good helper methods may contain different code or the exact same code (separate methods are used to easily evaluate tool results as "True Positives" or "False Positives").

- Helper methods for the bad code are named "helperBad."

- Ideally, helper methods would be specific to an individual secondary good method 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 method named "helperGood" is used.

- Helper methods are private methods when possible.

- In multi-file test cases, helper methods may be in the primary file or in the other, non-primary files.

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

```
^helperBad$

^helperGood(G2B|B2G)?\d*$
```

4.1.2.2 Source and Sink Methods

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

The following items describe source and sink methods further:

- Bad source and sink methods are generally named "badSource" and "badSink."

- Good source methods are generally named "goodG2BSource," "goodG2B1Source," "goodB2GSource," "goodB2G2Source," etc.

- Good sink methods are generally named "goodG2BSink," "goodG2B1Sink," "goodB2GSink," "goodB2G2Sink," etc.

- In multi-file test cases, source and sink methods may be defined in the primary file or in the other, non-primary files.

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

```
^badSource$

^badSink$
```

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

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

4.2 Class-Based Flaw Test Cases

The design of test cases for 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 method. 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 class:

- Is located in a file that ends in _bad (before the extension). For example, "CWE581_Object_ Model_Violation__hashCode_01_bad.java."
- Contains a required bad method with a signature like the bad method in a test case for a non-class-based flaw. This method makes use of the bad class for this test case to exercise the flaw being tested.
    - This is a public method in the class.
- The class is based on the file name (as required by the Java compiler). The class inherits AbstractTestCaseClassIssueBad.
- Has a main method that calls the bad method. Like the main methods in test cases for non-class-based flaws, this method is only used for testing or building separate archives for the test case.

Good File for Class-Based Flaws

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

- Is located in a file that ends in "_good1" (before the extension). For example, "CWE581_Object_ Model_Violation__hashCode_01_good1.java." Future versions of the test cases may include additional good files with names containing "_good2," "_good3," etc.
- Contains a required primary good method named "good" with a signature like the good method in a test case for a non-class-based flaw. Like the primary good method in a test case for a non-class-based flaw, this method only calls the secondary good method in this file.
    - This is a public method in the class.

- Contains at least one required secondary good method named “good1” to match the file name (currently, only “good1” method names are used, but future versions of the test cases may use methods “good2,” “good3,” etc.). The signature for this method is like the signature of a secondary good method in a test case for a non-class-based flaw. This secondary good method makes use of the class in this file to exercise the non-flawed construct being tested.
    - This is a private method in the class contained in the file.
- The class in the file is named based on the file name (as required by the Java compiler). The class inherits AbstractTestCaseClassIssueGood.
- Has a main method that calls the primary good method. Like the main methods in test cases for non-class-based flaws, this method is only used for testing or building separate archives for the test case.

4.3 Abstract Method Test Cases

A few test cases, like ones using flow variant 81, make use of abstract methods. 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.

An abstract method Data Flow test case contains five files:

1. Base file – This file defines an abstract base class and declares an abstract method, named “action,” within the base. The file name contains the string “base” and is a Java source file.
2. Root file – This file contains the implementations for the bad and good methods. The file name contains the letter ‘a’ as a sub-file identifier and is a Java source file.
3. Bad implementation file – This file implements the “action” method for the bad class, contains the string “bad” as a sub-file identifier, and is a Java source file.
4. GoodG2B implementation file - This file implements the “action” method 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 Java source file.
5. GoodB2G implementation file - This file implements the “action” method 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 Java source file.

As an example, the files for the CWE369_Divide_by_Zero__int_File_divide_81 test case are as follows:

- CWE369_Divide_by_Zero__int_File_divide_81_base.java
- CWE369_Divide_by_Zero__int_File_divide_81a.java
- CWE369_Divide_by_Zero__int_File_divide_81_bad.java
- CWE369_Divide_by_Zero__int_File_divide_81_goodG2B.java
- CWE369_Divide_by_Zero__int_File_divide_81_goodB2G.java

4.4 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.

- Some bad-only test cases contain an empty good method.

- Some bad-only test cases include good methods that contain the flawed construct within unreachable code.

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 method are provided to execute the test cases.

The following sections 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 *~src\testcasesupport* directory.

Abstract Class test case support files:

- *AbstractTestCase.java* – This file contains the base class which non-Servlet, non-class-based test cases extend. It contains abstract primary good and bad methods, along with implementations of common logic for running the test case.
- *AbstractTestCaseBadOnly.java* – This file contains the base class which bad-only based test cases extend. It contains an abstract primary bad method as well as an implementation of common logic for running the test case.
- *AbstractTestCaseBase.java* – This file contains the base class which all non-Servlet based test cases extend. It contains an abstract method for running the test case as well an implementation of a main method.
- *AbstractTestCaseClassIssue.java* - This file contains a base class for test cases that have flaws outside of a good or bad method and which are part of the class itself. It creates a protected object for a bad and good class as well as implementation of common logic for running the test case.
- *AbstractTestCaseClassIssueBad.java* – This file contains the base class which the bad class for class-based test cases extends. It contains the abstract primary bad method along with implementations of common logic for running the test case.
- *AbstractTestCaseClassIssueGood.java* – This file contains the base class which the good class for class-based test cases extends. It contains the abstract primary good method along with implementations of common logic for running the test case.
- *AbstractTestCaseServlet.java* – This file contains the base class which Servlet test cases extend. It contains abstract primary good and bad methods along with implementations of common logic for running the test case.
- *AbstractTestCaseServletBadOnly.java* – This file contains the base class which bad-only Servlet-based test cases extend. It contains an abstract primary bad method as well as an implementation of common logic for running the test case.
- *AbstractTestCaseServletBase.java* - This file contains the base class which all Servlet based test cases extend. It contains an abstract method for running the test case as well an implementation of a main method.

Input/Output related support files:

- *IO.java* – This file contains several methods used by the test cases to print various types of data to the console. Test cases use the methods in this file instead of calling console output methods directly in order to prevent incidental issue reports from analysis tools for "inappropriate logging" or "possible data disclosure." This file also contains a method to obtain a database connection and several methods and static variables used in Control Flow test cases.

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.

For example, a number of test cases for CWE-690 (Unchecked Return Value to NULL Pointer Dereference) use the class CWE690_NULL_Deref_From_Return__Class_Helper, which provides a non-system method which may return NULL. The file is located in the *~\src\testcases\CWE690_NULL_Deref_From_Return* directory. Because this file is used by multiple test cases, it is considered a support file rather than a "test case file." More importantly, this file does not contain the target flaw for the test cases.

5.3 Main Class Support Files

Support files are also provided to test an individual CWE entry for both non-Servlet and Servlet test cases. These files, called Main.java and ServletMain.java, are auto-generated and are included with each CWE entry (such as in the *~src\testcases\CWE23_Relative_Path_Traversal* 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 Java, 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.java file and a ServletMain.java file. These files can be used to compile and test all of the test cases contained within that subdirectory.

Each file is described below:

- *Main.java* – This file contains a "main" method that calls each non-Servlet test case and calls the "runTest" method on the object. The runTest method calls the test case's primary good and primary bad methods.

- *ServletMain.java* – This file contains a "doGet" method that indirectly creates an object for each Servlet test case and calls the "runTest" method on the object. The runTest method calls the test case's primary good and bad methods.

See Section 7.1 for details on how to update Main.java and ServletMain.java 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 testing was done using versions shown in parenthesis):

- Microsoft Windows platform (Windows 7)
- Oracle Java Development Kit (JDK) (Version 6)
- Apache Ant (version 1.8.4)
- Python for Windows (version 3.2.3)

In addition, the PATH environment variable must be set to properly point to the locations of the executables for Java, Ant, and Python.

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

6.2 Compiling Test Cases per CWE Entry

The test cases can be compiled so that a separate WAR file is generated for each CWE entry, with a few exceptions. This is accomplished by running "ant" in the directory for that CWE entry (such as in the *~src\testcases\CWE476_NULL_Pointer_Dereference* directory to create the file "~src\testcases\CWE476_NULL_Pointer_Dereference.war."

In order to automate the process of compiling the individual test cases in each CWE entry's directory, the Python script named "run_analysis_example_tool.py" can be executed. This script will go to each CWE entry directory and run "ant" 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.

6.2.1.1 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 190 are broken up into the following subdirectories:

- *~src\testcases\CWE190_Integer_Overflow\s01*
- *~src\testcases\CWE190_Integer_Overflow\s02*
- *~src\testcases\CWE190_Integer_Overflow\s03*
- *~src\testcases\CWE190_Integer_Overflow\s04*
- *~src\testcases\CWE190_Integer_Overflow\s05*

Each subdirectory contains a build.xml file that can be used to compile all of the test case files located within that directory using "ant." The generated WAR file contains the subdirectory number in its file name. For example, running "ant" in the *~src\testcases\CWE190_Integer_Overflow\s01* directory will compile the test cases in that directory into "~src\testcases\CWE190_Integer_Overflow_s01.war."

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 non-Servlet Test Case

This section describes commands to compile and run non-Servlet test cases. In each non-Servlet test case, a main method exists that contains a call to the primary good method for the test case, followed by a call to the primary bad method.

All commands should be run in the *~src* directory under the directory to which the test cases were extracted.

6.3.1.1 Non-split CWE Directories

The following example command will compile a single file test case, contained within a non-split directory, into appropriate .class files alongside the source files.

```
javac -cp ..\lib\servlet-api.jar;..\lib\commons-lang-
2.5.jar;..\lib\commons-codec-1.5.jar;..\lib\javamail-1.4.4.jar
testcasesupport\IO.java testcasesupport\AbstractTestCaseBase.java
testcasesupport\AbstractTestCase.java
testcasesupport\AbstractTestCaseBadOnly.java
testcasesupport\AbstractTestCaseServletBase.java
testcasesupport\AbstractTestCaseServlet.java
testcasesupport\AbstractTestCaseServletBadOnly.java
testcasesupport\AbstractTestCaseClassIssue.java
testcasesupport\AbstractTestCaseClassIssueBad.java
testcasesupport\AbstractTestCaseClassIssueGood.java
testcases\CWE78_OS_Command_Injection\CWE78_OS_Command_Injection__consol
e_readLine_01.java
```

The following example command will run the test case compiled above.

```
java -cp ..\lib\servlet-api.jar;..\lib\commons-lang-
2.5.jar;..\lib\commons-codec-1.5.jar;..\lib\javamail-1.4.4.jar;.
testcases.CWE78_OS_Command_Injection.CWE78_OS_Command_Injection__consol
e_readLine_01
```

The following example command will compile a multiple file test case, contained within a non-split directory, into appropriate .class files alongside the source files.

```
javac -cp ..\lib\servlet-api.jar;..\lib\commons-lang-
2.5.jar;..\lib\commons-codec-1.5.jar;..\lib\javamail-1.4.4.jar
testcasesupport\IO.java testcasesupport\AbstractTestCaseBase.java
testcasesupport\AbstractTestCase.java
testcasesupport\AbstractTestCaseBadOnly.java
testcasesupport\AbstractTestCaseServletBase.java
testcasesupport\AbstractTestCaseServlet.java
testcasesupport\AbstractTestCaseServletBadOnly.java
testcasesupport\AbstractTestCaseClassIssue.java
testcasesupport\AbstractTestCaseClassIssueBad.java
testcasesupport\AbstractTestCaseClassIssueGood.java
testcases\CWE78_OS_Command_Injection\CWE78_OS_Command_Injection__consol
e_readLine_54*.java
```

The following example command will run the test case compiled above. Note that the "main" for the test case is contained in the "a" file for the test case.

```
java -cp ..\lib\servlet-api.jar;..\lib\commons-lang-
2.5.jar;..\lib\commons-codec-1.5.jar;..\lib\javamail-1.4.4.jar;.
testcases.CWE78_OS_Command_Injection.CWE78_OS_Command_Injection__consol
e_readLine_54a
```

6.3.1.2 Split CWE Directories

The following example command will compile a single file test case, contained within a subdirectory of a split CWE directory, into appropriate .class files alongside the source files.

```
javac -cp ..\lib\servlet-api.jar;..\lib\commons-lang-
2.5.jar;..\lib\commons-codec-1.5.jar;..\lib\javamail-1.4.4.jar
testcasesupport\IO.java testcasesupport\AbstractTestCaseBase.java
testcasesupport\AbstractTestCase.java
testcasesupport\AbstractTestCaseBadOnly.java
testcasesupport\AbstractTestCaseServletBase.java
testcasesupport\AbstractTestCaseServlet.java
testcasesupport\AbstractTestCaseServletBadOnly.java
testcasesupport\AbstractTestCaseClassIssue.java
testcasesupport\AbstractTestCaseClassIssueBad.java
testcasesupport\AbstractTestCaseClassIssueGood.java
testcases\CWE190_Integer_Overflow\s01\CWE190_Integer_Overflow__byte_con
sole_readLine_add_01.java
```

The following example command will run the test case compiled above.

```
java -cp ..\lib\servlet-api.jar;..\lib\commons-lang-
2.5.jar;..\lib\commons-codec-1.5.jar;..\lib\javamail-1.4.4.jar;.
testcases.CWE190_Integer_Overflow.s01.CWE190_Integer_Overflow__byte_con
sole_readLine_add_01
```

The following example command will compile a multiple file test case, contained within a subdirectory of a split CWE directory, into appropriate .class files alongside the source files.

```
javac -cp ..\lib\servlet-api.jar;..\lib\commons-lang-
2.5.jar;..\lib\commons-codec-1.5.jar;..\lib\javamail-1.4.4.jar
testcasesupport\IO.java testcasesupport\AbstractTestCaseBase.java
testcasesupport\AbstractTestCase.java
testcasesupport\AbstractTestCaseBadOnly.java
testcasesupport\AbstractTestCaseServletBase.java
testcasesupport\AbstractTestCaseServlet.java
testcasesupport\AbstractTestCaseServletBadOnly.java
testcasesupport\AbstractTestCaseClassIssue.java
testcasesupport\AbstractTestCaseClassIssueBad.java
testcasesupport\AbstractTestCaseClassIssueGood.java
testcases\CWE190_Integer_Overflow\s01\CWE190_Integer_Overflow__byte_con
sole_readLine_add_66*.java
```

The following example command will run the test case compiled above. Note that the “main” for the test case is contained in the “a” file for the test case.

```
java -cp ..\lib\servlet-api.jar;..\lib\commons-lang-
2.5.jar;..\lib\commons-codec-1.5.jar;..\lib\javamail-1.4.4.jar;.
testcases.CWE190_Integer_Overflow.s01.CWE190_Integer_Overflow__byte_con
sole_readLine_add_66a
```

6.3.2 Building and Running a Servlet Test Case

In order to compile each test case separately, the Servlet test cases inherit the javax.servlet.http.HttpServlet class. The process for building and running a Servlet test case requires multiple steps and use of a Servlet container. Due to the number of steps, an example of how to build and run a Servlet test case using the Eclipse J2EE Preview Server is detailed in Appendix F.

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 Build Files

The Java test case archive contains a Python script that can be used to update the build files if changes are made to the set of test cases to be analyzed.

> *create_per_cwe_files.py* – Running this script will update the Main.java, ServletMain.java, and web.xml files in each CWE entry directory that allow for building the test cases for that CWE entry into a separate WAR file.

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 method with the word “bad” in its name (such as bad(), badSource(), or badSink()) or in a class that contains the word “bad” in its name (such as CWE581_Object_Model_Violation__hashCode_01_bad). 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 methods 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 method 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 method or class with the word “good” in its name. An incorrect report of the target flaw type in a good method 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 methods that contain a non-flawed construct. When a test case has more than one secondary method, a test case user may want to determine in which secondary good method(s) a tool reported False Positives and in which secondary good method(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 methods where tool results are reported. The “source” and “sink” methods can be associated with the secondary good method from which they are called (for example, the method goodB2GSource or goodB2GSink can be associated with the secondary good method 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 methods in all test cases. Specifically, as detailed in Section 4.1.2.1, good helper methods are not specific to the secondary good methods in a test case. Therefore, in a test case with more than one secondary good method and one or more False Positive results in a good helper method, the

False Positive(s) cannot be easily associated with secondary good method(s) and True Negatives cannot be determined, either.[3]

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.

[3] This association cannot be made based solely on method 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	Java Test Cases
15	External Control of System or Configuration Setting	444
23	Relative Path Traversal	444
36	Absolute Path Traversal	444
78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	444
80	Improper Neutralization of Script-Related HTML Tags in a Web Page (Basic XSS)	666
81	Improper Neutralization of Script in an Error Message Web Page	333
83	Improper Neutralization of Script in Attributes in a Web Page	333
89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	2220
90	Improper Neutralization of Special Elements used in an LDAP Query ('LDAP Injection')	444
111	Direct Use of Unsafe JNI	1
113	Improper Neutralization of CRLF Sequences in HTTP Headers ('HTTP Response Splitting')	1332
114	Process Control	17
129	Improper Validation of Array Index	2664
134	Uncontrolled Format String	666
190	Integer Overflow or Wraparound	2553
191	Integer Underflow (Wrap or Wraparound)	1702
193	Off-by-one Error	51
197	Numeric Truncation Error	1221
209	Information Exposure Through an Error Message	34
226	Sensitive Information Uncleared Before Release	17
248	Uncaught Exception	1
252	Unchecked Return Value	17
253	Incorrect Check of Function Return Value	17
256	Plaintext Storage of a Password	37
259	Use of Hard-coded Password	111
315	Plaintext Storage in a Cookie	37
319	Cleartext Transmission of Sensitive Information	370
321	Use of Hard-coded Cryptographic Key	37
325	Missing Required Cryptographic Step	34
327	Use of a Broken or Risky Cryptographic Algorithm	34
328	Reversible One-Way Hash	51
329	Not Using a Random IV with CBC Mode	17
336	Same Seed in PRNG	17
338	Use of Cryptographically Weak PRNG	34
369	Divide By Zero	1850
378	Creation of Temporary File With Insecure Permissions	17
379	Creation of Temporary File in Directory with Incorrect Permissions	17
382	J2EE Bad Practices: Use of System.exit()	34
383	J2EE Bad Practices: Direct Use of Threads	16

390	Detection of Error Condition Without Action	34
395	Use of NullPointerException Catch to Detect NULL Pointer Dereference	17
396	Declaration of Catch for Generic Exception	34
397	Declaration of Throws for Generic Exception	4
398	Indicator of Poor Code Quality	137
400	Uncontrolled Resource Consumption ('Resource Exhaustion')	1460
404	Improper Resource Shutdown or Release	5
459	Incomplete Cleanup	34
470	Use of Externally-Controlled Input to Select Classes or Code ('Unsafe Reflection')	444
476	NULL Pointer Dereference	198
477	Use of Obsolete Functions	68
478	Missing Default Case in Switch Statement	17
481	Assigning instead of Comparing	17
482	Comparing instead of Assigning	17
483	Incorrect Block Delimitation	19
484	Omitted Break Statement in Switch	17
486	Comparison of Classes by Name	17
491	Public cloneable() Method Without Final ('Object Hijack')	1
499	Serializable Class Containing Sensitive Data	1
500	Public Static Field Not Marked Final	1
506	Embedded Malicious Code	116
510	Trapdoor	66
511	Logic/Time Bomb	51
523	Unprotected Transport of Credentials	17
526	Information Exposure Through Environmental Variables	34
533	Information Exposure Through Server Log Files	17
534	Information Exposure Through Debug Log Files	17
535	Information Exposure Through Shell Error Message	17
539	Information Exposure Through Persistent Cookies	17
546	Suspicious Comment	85
549	Missing Password Field Masking	17
561	Dead Code	1
563	Unused Variable	218
566	Authorization Bypass Through User-Controlled SQL Primary Key	37
568	finalize() Method Without super.finalize()	2
570	Expression is Always False	16
571	Expression is Always True	16
572	Call to Thread run() instead of start()	17
579	J2EE Bad Practices: Non-serializable Object Stored in Session	1
580	clone() Method Without super.clone()	1
581	Object Model Violation: Just One of Equals and Hashcode Defined	2
582	Array Declared Public, Final, and Static	1
584	Return Inside Finally Block	17
585	Empty Synchronized Block	2
586	Explicit Call to Finalize()	17
597	Use of Wrong Operator in String Comparison	17
598	Information Exposure Through Query Strings in GET Request	17
600	Uncaught Exception in Servlet	1
601	URL Redirection to Untrusted Site ('Open Redirect')	333
605	Multiple Binds to the Same Port	17
606	Unchecked Input for Loop Condition	444
607	Public Static Final Field References Mutable Object	1

609	Double-Checked Locking	2
613	Insufficient Session Expiration	17
614	Sensitive Cookie in HTTPS Session Without 'Secure' Attribute	17
615	Information Exposure Through Comments	17
617	Reachable Assertion	34
643	Improper Neutralization of Data within XPath Expressions ('XPath Injection')	444
667	Improper Locking	1
674	Uncontrolled Recursion	2
681	Incorrect Conversion between Numeric Types	51
690	Unchecked Return Value to NULL Pointer Dereference	296
698	Execution After Redirect (EAR)	17
759	Use of a One-Way Hash without a Salt	17
760	Use of a One-Way Hash with a Predictable Salt	17
764	Multiple Locks of a Critical Resource	2
765	Multiple Unlocks of a Critical Resource	2
772	Missing Release of Resource after Effective Lifetime	2
775	Missing Release of File Descriptor or Handle after Effective Lifetime	2
789	Uncontrolled Memory Allocation	1571
832	Unlock of a Resource that is not Locked	2
833	Deadlock	6
835	Loop with Unreachable Exit Condition ('Infinite Loop')	6

Table 4 – CWE Entries in 2012 Test Cases (Java)

Appendix B: CWE/SANS Top 25 Coverage

Table 5 shows the degree to which the Java 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**	**Java**
1	CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	CWE-89	2220
2	CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	CWE-78	444
3	CWE-120: Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')	None (Buffer Handling issues are covered in the related C/C++ test cases)	-
4	CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	CWE-80: Improper Neutralization of Script-Related HTML Tags in a Web Page (Basic XSS), CWE-81: Improper Neutralization of Script in an Error Message Web Page, CWE-83: Improper Neutralization of Script in Attributes in a Web Page	1332
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	148
8	CWE-311: Missing Encryption of Sensitive Data	CWE-315: Plaintext Storage in a Cookie, CWE-319: Cleartext Transmission of Sensitive Information	407
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	None (Covered in the related C/C++ test cases)	-

2011 CWE/SANS Top 25		CAS Test Cases	
Rank	CWE Entry	CWE Entry / Entries	Java
11	CWE-250: Execution with Unnecessary Privileges	None (Design issue which does not fit into CAS Test Case structure)	-
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	888
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	None (Covered in the related C/C++ test cases)	-
19	CWE-327: Use of a Broken or Risky Cryptographic Algorithm	CWE-327	34
20	CWE-131: Incorrect Calculation of Buffer Size	None (Does not fit into CAS Test Case structure for Java)	-
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')	CWE-601	333
23	CWE-134: Uncontrolled Format String	CWE-134	666
24	CWE-190: Integer Overflow or Wraparound	CWE-190, CWE-191: Integer Underflow (Wrap or Wraparound)	4255
25	CWE-759: Use of a One-Way Hash without a Salt	CWE-759	17

Table 5 – 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 Java test cases, including a brief description. Due to design constraints, all flaw types do not contain test cases for each flow variant.

Flow Variant	Flow Type	Description
01	None	Baseline – Simplest form of the flaw
02	Control	if(true) and if(false)
03	Control	if(5==5) and if(5!=5)
04	Control	if(PRIVATE_STATIC_FINAL_TRUE) and if(PRIVATE_STATIC_FINAL_FALSE)
05	Control	if(privateTrue) and if(privateFalse)
06	Control	if(PRIVATE_STATIC_FINAL_FIVE==5) and if(PRIVATE_STATIC_FINAL_FIVE!=5)
07	Control	if(privateFive==5) and if(privateFive!=5)
08	Control	if(privateReturnsTrue()) and if(privateReturnsFalse())
09	Control	if(IO.STATIC_FINAL_TRUE) and if(IO.STATIC_FINAL_FALSE)
10	Control	if(IO.staticTrue) and if(IO.staticFalse)
11	Control	if(IO.staticReturnsTrue()) and if(IO.staticReturnsFalse())
12	Control	if(IO.staticReturnsTrueOrFalse())
13	Control	if(IO.STATIC_FINAL_FIVE==5) and if(IO.STATIC_FINAL_FIVE!=5)
14	Control	if(IO.staticFive==5) and if(IO.staticFive!=5)
15	Control	switch(6) and switch(7)
16	Control	while(true)
17	Control	for loops
21	Control	Flow controlled by value of a private variable. All methods contained in one file.
22	Control	Flow controlled by value of a public static variable. Sink methods are in a separate file from sources.
31	Data	Data flow using a copy of data within the same method
41	Data	Data passed as an argument from one method to another in the same class
42	Data	Data returned from one method to another in the same class
45	Data	Data passed as a private class member variable from one method to another in the same class
51	Data	Data passed as an argument from one method to another in different classes in the same package
52	Data	Data passed as an argument from one method to another to another in three different classes in the same package
53	Data	Data passed as an argument from one method through two others to a fourth; all four methods are in different classes in the same package
54	Data	Data passed as an argument from one method through three others to a fifth; all five methods are in different classes in the same package
61	Data	Data returned from one method to another in different classes in the same package
66	Data	Data passed in an array from one method to another in different classes in the same package
67	Data	Data passed in a class from one method to another in different classes in the same package

Flow Variant	Flow Type	Description
68	Data	Data passed as a member variable in the “a” class from one method to another in different classes in the same package
71	Data	Data passed as an Object reference argument from one method to another in different classes in the same package
72	Data	Data passed in a Vector from one method to another in different classes in the same package
73	Data	Data passed in a LinkedList from one method to another in different classes in the same package
74	Data	Data passed in a HashMap from one method to another in different classes in the same package
75	Data	Data passed in a serialized object from one method to another in different classes in the same package
81	Data	Data passed in an argument to an abstract method called via a reference

Table 6 – Test Case Flow Variants

Appendix D : Bad-Only Test Cases

CWE Entry ID	CWE Entry Name	Functional Variants	Flow Variants
111	Direct Use of Unsafe JNI	*	*
383	J2EE Bad Practices: Direct Use of Threads	*	*
506	Embedded Malicious Code	email file_transfer_connect_tcp file_transfer_listen_tcp screen_capture	*
510	Trapdoor	network_connection network_listen	*

* = Applicable to all variants.

Table 7 – Java 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 Java:

CWE Entry ID	CWE Entry Name
15	External Control of System or Configuration Setting
197	Numeric Truncation Error
226	Sensitive Information Uncleared Before Release
459	Incomplete Cleanup
526	Information Exposure Through Environmental Variables
539	Information Exposure Through Persistent Cookies
667	Improper Locking

Table 8 – CWEs Added in Juliet Test Suite v1.2 for Java

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 Java:

CWE Entry ID	CWE Name
180	Incorrect Behavior Order: Validate Before Canonicalize
330	Use of Insufficiently Random Values
489	Leftover Debug Code
497	Exposure of System Data to an Unauthorized Control Sphere
514	Covert Channel
547	Use of Hard-coded, Security-relevant Constants
665	Improper Initialization
784	Reliance on Cookies without Validation and Integrity Checking in a Security Decision

Table 9 – CWEs Removed in Juliet Test Suite v1.2 for Java

The following flow variant was removed in Juliet Test Suite v1.2 for Java:

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 Java

The number of flaw types for the following CWEs changed in Juliet Test Suite v1.2 for Java. 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
80	Improper Neutralization of Script-Related HTML Tags in a Web Page (Basic XSS)	12	18	6
113	Improper Neutralization of CRLF Sequences in HTTP Headers ('HTTP Response Splitting')	48	36	-12
129	Improper Validation of Array Index	58	72	14
193	Off-by-one Error	1	3	2
327	Use of a Broken or Risky Cryptographic Algorithm	1	2	1
328	Reversible One-Way Hash	1	3	2
382	J2EE Bad Practices: Use of System.exit()	1	2	1
390	Detection of Error Condition Without Action	3	2	-1
398	Indicator of Poor Code Quality	4	9	5
400	Uncontrolled Resource Consumption ('Resource Exhaustion')	29	40	11
404	Improper Resource Shutdown or Release	3	5	2
476	NULL Pointer Dereference	4	7	3
499	Serializable Class Containing Sensitive Data	2	1	-1
506	Embedded Malicious Code	7	8	1
568	finalize() Method Without super.finalize()	1	2	1
570	Expression is Always False	6	16	10
571	Expression is Always True	6	16	10
581	Object Model Violation: Just One of Equals and Hashcode Defined	1	2	1
760	Use of a One-Way Hash with a Predictable Salt	12	1	-11
775	Missing Release of File Descriptor or Handle after Effective Lifetime	1	2	1

Table 11 – Flaw Count Changes in Juliet Test Suite v1.2 for Java

Appendix F : Running a Servlet Test Case in Eclipse

The instructions below describe how to run an individual Servlet test case in Eclipse using the J2EE Preview Server. This process can be used for testing or debugging a Servlet test case, but is not needed when compiling or analyzing the test cases.

These instructions were created using the Eclipse version named Indigo Service Release 2. Details may vary for other versions.

1. Create a new project
    a. File-> New-> Project. Expand Web and select Dynamic Web Project.
    b. Click Next.
    c. For Project Name, use "TestCases"
    d. If you have nothing in the Target Runtime dropdown, or you do NOT have J2EEPreview as a selectable option, click New.
    e. Expand Basic, select J2EEPreview and click Finish.
    f. Select J2EE Preview as the Target Runtime.
    g. Click Finish.
2. Add the support files
    a. Select File->Import
    b. Select General->File System
    c. Click Next
    d. In From Directory, set it to the Java test case's source directory, such as "C:\testcases\Java\src"
    e. Expand src
    f. Check the checkbox next to "testcasesupport" folder in the left hand pane
    g. Click on Filter Types button, and select only .java files.
    h. In "Into Folder," select the project src folder, such as "TestCases/src"
    i. Click Finish
3. Add the library files
    a. Select File->Import
    b. Select General->File System
    c. Click Next
    d. In From Directory, set it to the Java test case directory, such as "C:\testcases\Java\"
    e. Expand Java
    f. Check the checkbox next to "lib" folder in the left hand pane
    g. Click on Filter Types button, and select only .jar files.
    h. In "Into Folder," select the project folder, such as "TestCases"
    i. Click Finish

4. Add references to lib files
    a. Right click the "TestCases" project, select Properties
    b. Select Java Build Path, then the Libraries tab
    c. Click Add JARs
    d. Browse to TestCases\lib and select commons-lang-2.5.jar, commons-codec-1.5.jar, and javamail-1.4.4.jar
    e. Click OK
    f. Click OK Again
5. Add a test case you want to test
    a. Select File->Import
    b. Select General->File System
    c. Click Next
    d. In From Directory, set it to the Java test case's source directory, such as "C:\testcases\Java\src"
    e. Expand src
    f. Expand testcases
    g. Check the checkbox next to the CWE folder that contains the test case you want to test
    h. Click on Filter Types button, and select only .java files.
    i. In "Into Folder," select the project src folder, such as "TestCases/src"
    j. Click Finish
6. Associate server to this project
    a. Right-click the project, select New->Other. Expand Server. Select Server. Click Next.
    b. Expand Basic, select "J2EE Preview"
    c. Click Next.
    d. Select your project and click Add.
    e. Click Finish.
7. Run Servlet
    a. Right-click the test case you want to run and select New->Servlet. This step will add it to the web.xml.
    b. Leave all the defaults, and click Finish.
    c. Right-click the test case you want to try and run, select Run As, then Run on Server.
    d. The first time you run, you may be asked to select which server to use. Make sure you select the J2EE preview server.
    e. Click Finish.
    f. Click unblock if a windows firewall message pops up.
    g. If you get an error, go to the Servers tab on the bottom, right click the J2EE Preview server and click stop.
    h. Try to rerun the test case.