The Juliet 1.1 C/C++ and Java Test Suite

Tim Boland and Paul E. Black
National Institute of Standards and Technology

Juliet Test Suite 1.1 offers test cases for assessing the effectiveness of static analyzers and other software-assurance tools.

Juliet Test Suite 1.1 is a collection of C/C++ and Java programs with known flaws comprising 57,099 test cases in C/C++ and 23,957 test cases in Java. Most of the C/C++ cases are in C; C++ is used when the flaw occurs only in C++. Each program or test case consists of one or two pages of code, and most include similar but unflawed code to test discrimination.

Juliet covers 181 different kinds of flaws documented by common weakness enumeration (CWE; http://cwe.mitre.org), including authentication and access control, buffer handling, code quality, control-flow management, encryption and randomness, error handling, file handling, information leaks, initialization and shutdown, injection, and pointer and reference handling. Juliet offers examples of each flaw in simple code as well as cases in which the flaw is embedded in variations of three dozen different control flow- and data-flow patterns.

The test cases are synthetic, that is, they were created as examples with well-characterized weaknesses. Each case targets only one flaw. As a result, the cases have a much simpler structure than most weaknesses in production code, and users shouldn't extrapolate statistics, such as rate of occurrence or severity.

Developed by the National Security Agency's Center for Assured Software, the test suite is in the public domain and isn't subject to copyright protection. The test cases have undergone limited review to verify that each contains the weakness and flow variation it purports to have, but there is no guarantee. Every test case has been successfully compiled.

The test suite includes C/C++ and Java User Guides explaining many of the details of the structure and possible uses of Juliet. Users can download the C/C++ or Java portions of the Juliet Test Suite from the top of the Test Suites page of the National Institute of Standards and Technology (NIST) Software Assurance Metrics and Tool Evaluation (SAMATE) Reference Dataset (SRD; http://samate.nist.gov/SRD/testsuite.php).

BACKGROUND

Juliet is SRD's tenth major contribution (the tenth letter of the International Radiotelephony Spelling Alphabet is "Juliet"). The original version, 1.0, released in December 2010, is available in the SRD as individual cases, which makes browsing the cases easier. Users also can download the full C/C++ or full Java portions of version 1.0. The current version, 1.1, will be added to the SRD as individual cases.

Juliet's development team created test cases for selected flaws based on several factors, including the team's experience, the flaw's importance or severity, and its frequency of occurrence. The test cases cover 14 of the 2011 CWE/SANS (SysAdmin, Audit, Network, Security) Top 25 Most Dangerous Programming Errors (http://cwe.mitre.org/top25). The remaining 11 flaws are design issues, such as CWE-862 Missing Authorization, and CWE-250 Execution with Unnecessary Privileges, which don't fit Juliet's test case approach.

TEST CASE STRUCTURE

Each test case's name follows a specific format and its code has a particular structure.

Name

Each test case resides in one or more files with names such as "CWE134_Uncontrolled_Format_String__char_file_printf_22a.c." The file name has the following components: a CWE number (134), a short name (Uncontrolled_Format_

Published by the IEEE Computer Society

String), a functional variant (char_file_printf), a two-digit flow structure number (22), an optional subfile indicator (a), and the appropriate extension (.c).

Functional variants might name data types, library functions, or structures. Java servlets have "Servlet" in the functional variant. Windows-specific test cases contain "w32" in the functional variant.

Flow structure numbers indicate the type of data or control flow used, for example, loop; data flow; local control flow; constant in conditional; data passing involving functions, methods, or classes; data type; container; or combined control and data flow. Test cases with the same flow structure number have the same type of data or control flow. Flow structure "01" indicates baseline or the simplest instance.

A test case can comprise one source code file or multiple files. For example, test case CWE476_NULL_Pointer_Dereference__char_01 is contained in one source code file, whereas test case CWE23_Relative_Path_Traversal__wchar_t_connect_socket_w32CreateFile_54 has five files—..._54a.c, ..._54b.c, through ..._54e.c—that constitute one test case. Some cases use "bad" or "good" as subfile indicators for flawed or unflawed code.

Code

Although each test case targets only one flaw, other incidental flaws might be present. For example, CWE489_Leftover_Debug_Code__Servlet_01.java focuses on leftover debug code, but it also includes a CWE-259 Hardcoded Password weakness. Many cases have infeasible (unreachable or "dead") code with or without flaws.

Each source file begins with a block of comments with the test case's name, the basis, and the variants in the file. Following the comments is a "bad" portion containing the flaw, one or more good portions that don't contain the flaw, and a "main" function that can be used to compile the test case as a stand-alone program.

In some cases, the bad and good portions are in separate files. The good portions provide one or more functions or methods having behavior similar to the bad portion, but without the flaw. Bad code is an opportunity for a tool to identify a flaw. Good code exercises the tool's ability to distinguish between flawed and unflawed code. For C/C++ test cases, users can choose preprocessor directives for only the bad parts in the code, only the good parts, or both.

The Juliet test suite focuses on functions available on the underlying platform, not on third-party libraries or frameworks.

Users can compile all test cases in one huge program, all cases for one CWE, or each test case individually. The user guides offer instructions on how to accomplish each of these compilations.

Readers can access Juliet 1.0 test case examples, which are structurally identical, at http://samate.nist.gov/SRD. For instance, `memcpy()` overflows a `char` array in a `struct` in test case CWE121_Stack_Based_Buffer_Overflow__char_type_overrun_memcpy_01 (http://samate.nist.gov/SRD/view_testcase.php?tID=2716). CWE114_Process_Control__basic_06, a Java example using `loadLibrary()` instead of `load()` and a path, is available at http://samate.nist.gov/SRD/view_testcase.php?tID=48995.

TEST SUITE STRUCTURE

The Juliet Test Suite focuses on functions available on the underlying platform, not on third-party libraries or frameworks. Test cases generally emphasize platform-neutral functions but contain some Windows-specific functions. The test cases were developed on the Microsoft Windows platform. Java test cases cover Java applications and servlets, but not applets or JavaServer Pages (JSPs). The test cases don't cover mobile applications or embedded code. They aren't meant to be an absolute measure of tool or software quality, to address complex data structures, or to address frequencies of production code flaws.

Using a custom tool, the development team generated most of the test cases from source files with the flaw. They created test cases for some types of weaknesses manually. There are more variations and types of weaknesses in C/C++ than in Java, hence the larger number of test cases for C/C++. The number of test cases per weakness varies widely, depending on the number of variations and code complexities.

The main C and Java directories contain utilities and miscellaneous files, including Python scripts for different compilations. Test cases are organized by weakness—C/testcases and Java/src/testcases have a subdirectory for each weakness containing source code files for test cases as well as a few auxiliary files for compiling. Some weakness subdirectories also contain supplemental files with extensions such as .so, .dll, and .project. A handful of auxiliary files, which reside in subdirectories C/testcasesupport or Java/src/testcasesupport, provide common functions.

The test cases strive to use the most specific CWE entry for the flaw of focus. For example, there are no test cases for CWE-119 Improper Restriction of Operations within the Bounds of a Memory Buffer; instead, the more specific CWE-121 Stack-Based Buffer Overflow is used. When the flaw doesn't correspond exactly to any CWE, the closest CWE is used.

POSSIBLE USES

There are several ways users can apply the Juliet Test Suite to help understand their software-assurance tools' capabilities. One use is analyzing the test cases as a single, large program, which indicates how a software-assurance tool performs on such programs. Because of the number of files and LOC, some tools might not be able to analyze all these test cases as a single program. Another use is to analyze separate test cases individually or in groups.

Because Juliet has thousands of separate test cases, users can select a particularly important set of flaws to examine; hence, studying a wide range of tools' capabilities is possible. Finally, users can select certain subsets of data and control flows to consider and compare across flaws to study the depth of the tools' analysis.

Because the test cases indicate where flaws occur, users can evaluate the reports' appropriateness semiautomatically. When users run a source code analysis tool on a test case, the desired result is for the tool to report one or more flaws of the target type in a function or method with "bad" in its name. A report of this type might be considered a true positive. If the tool doesn't report a flaw of the target type in a bad method, it might be considered a false negative. Ideally, the tool won't report flaws of the target type in a method with "good" in its name; a report of this type might be considered a false positive. Because flawed and similar unflawed code might be in infeasible code, users' policies on warnings about infeasible code must be taken into account.

Because test cases might contain flaws of nontarget types, users can ignore reports of flaws other than the target type.

Users can improve their security by finding and removing—or at least mitigating—program flaws. With Juliet, users can ask which static-analysis tools are most effective at finding flaws that are important to them. Other questions include how well do techniques such as memory layout randomization mitigate flaws in practice and what is wrong with calling thread run instead of start?

The Juliet Test Suite is a rich, structured resource to help users gauge the effectiveness of their software- and system-assurance methodology, bringing us one step closer to being able to measure security.

Disclaimer

The Juliet Test Suite is an experimental system. NIST assumes no responsibility whatsoever for its use by other parties and makes no guaranties, expressed or implied, about its quality, reliability, or any other characteristic.

Certain commercial equipment or materials are identified in this article to specify the experimental procedure adequately. Such identification is not intended to imply recommendation or endorsement by NIST, nor is it intended to imply that the materials or equipment identified are necssarily the best available for the purpose.

Tim Boland *is a computer scientist at the National Institute of Standards and Technology. Contact him at t.boland@nist.gov.*

Paul E. Black *is a computer scientist at the National Institute of Standards and Technology. Contact him at paul.black@nist.gov.*

Editor: Jeffrey Voas, National Institute of Standards and Technology; jeffrey.m.voas@gmail.com