National Institute of Standards and Technology Technical Note 1995
Natl. Inst. Stand. Technol. Tech. Note 1995, 36 pages (June 2018)
CODEN: NTNOEF

This publication is available free of charge from:
https://doi.org/10.6028/NIST.TN.1995

Abstract

The Juliet test suite is a systematic set of thousands of small test programs in C/C++ and Java, exhibiting over 100 classes of errors, such as buffer overflow, OS injection, hard-coded password, absolute path traversal, NULL pointer dereference, uncaught exception, deadlock, and missing release of resource. These test programs should be helpful in determining capabilities of software assurance tools, particularly static analyzers, in Unix, Microsoft Windows, and other environments. Juliet was developed by the National Security Agency's Center for Assured Software and first released in December 2010. It has been enhanced twice since then. Version 1.2 was released in May 2013 with a total of 86 864 test cases.

In the years after its release, many problems and deficiencies in Version 1.2 came to our attention. Released in October 2017, Version 1.3 fixes about fourteen systematic problems in Version 1.2 and adds tests for prefix and postfix increment integer overflow and decrement integer underflow. This technical note details the changes from Version 1.2 to 1.3. This note also lists known problems remaining in Juliet 1.3.

Key words

Buffer overflow; Bugs Framework (BF); Common Weakness Enumeration (CWE); cybersecurity; integer overflow; Juliet test suite; OS injection bugs; programming language test material; software assurance; software quality; static analysis; static source code analyzers.

Table of Contents

1. Introduction

We are pleased to announce Juliet 1.3, which was released in October 2017.

Juliet 1.3 consists of tens of thousands of small test programs in C/C++ and Java exhibiting over 100 classes of errors. It replaces Juliet 1.2. The C/C++ part contains 64 099 test cases and more than 100 000 files. The Java part contains 28 886 test cases and more than 46 000 files. Both parts also include files, scripts, headers, and other material for compiling the test cases, either as a single program per test case or as one program of all test cases in a given language. These cases should be useful in Unix, Microsoft Windows, and other environments. The Juliet test suite was originally developed by the National Security Agency's Center for Assured Software (CAS) and was first released in December 2010. We now refer to it as Juliet Version 1.0.

The C/C++ part of Juliet 1.0 comprised 45 324 test cases [1] covering 116 Common Weakness Enumeration (CWE) entries [2], and the Java part comprised 13 801 cases [3] covering 106 CWEs. The following year, Version 1.1 added a few additional CWEs and increased the total number of test cases to 81 056. To add methods for building test cases, Version 1.1.1 was released for the Java part.

Version 1.2 was released in May 2013 with a total of 86 864 test cases. A dozen CWEs were added, and during quality control review, CAS determined that test cases for the CWEs listed in Table 1 were invalid and removed them from the Java part [4].

Table 1. CWEs removed from the Java part of Version 1.2.

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

Similarly, Table 2 lists the CWEs that CAS determined were invalid and removed from the C/C++ part [5]. These are still available from the Software Assurance Reference Dataset (SARD) Test Suites page [6, 7] in Juliet Versions 1.1.

Table 2. CWEs removed from the C/C++ part of Version 1.2.

CWE	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

Flow variant 19, dead code after a return, was removed to reduce incidental dead code. (See the next section for an explanation of flow variants.) Two flow variants were added: 83, declaring class objects on the stack, and 84, declaring them in the heap. In addition, directories with many files were split into smaller subdirectories, so that no directory had more than 1000 files.

In the years since Version 1.2 was released, people using it reported unintentional problems they found and passed along suggestions for improvement. We received particularly extensive comments from Pascal Cuoq and André Maroneze. In 2016 one NIST researcher, Eric Trapnell collected much external and internal feedback and many notes, and we began to create a new version of Juliet to address the problems.

Juliet 1.3 fixes about two dozen systematic problems in Version 1.2. The fixes changed 21 552 files.

This technical note details the changes from Version 1.2. The next section, 1.1, briefly explains how the thousands of test cases in Juliet are organized, the case naming scheme, and the structure of each case. Section 1.2 is a very brief description of the fixes and changes. They are listed roughly in decreasing importance. Section 2 details each fix or change. In spite of all the changes, we know of many problems remaining in Juliet 1.3. Section 3 lists them and also lists suggestions that we did not take. Finally, Sec. 4 offers some thoughts about the future of Juliet and test suites in general.

1.1 The Organization of the Juliet Suite of Test Cases

The Juliet suite of test cases consists of two parts: test cases and supporting files for Java and test cases and supporting files for C and C++. Each part is available in two forms: a complete, structured, stand-alone suite and a suite of individual cases. These are available from the Software Assurance Reference Dataset (SARD) Test Suites page [6, 7].

The stand-alone suites include the CAS documentation for Version 1.2, shared support code and "include" files, means to compile the test cases (and scripts to update them if one adds or removes cases), and input files. Each CWE has its own subdirectory, e.g., `CWE338_Weak_PRNG` or `CWE764_Multiple_Locks`. CWEs with fewer than one thousand test case files contain all their test cases directly under its subdirectory. For CWEs with more

than one thousand files, the test cases are divided into subdirectories named s01, s02, etc. There is more information in the C/C++ or Java User Guides [4, 5].

In this document, we usually refer to just the CWE number, instead of the complete directory name, which includes the CWE name.

We use the Bugs Framework (BF) [8] in many instances for more clear and precise classification than is possible with CWEs.

Every Juliet test case is available as a separate test case in the SARD, with its own SARD ID number. For instance, `CWE80_XSS__Servlet_getParameter_Servlet_03.java` is 145277, and `CWE457_Use_of_Uninitialized_Variable__double_pointer_15.c` is 240543. The suites of individual cases, SARD test suites 108 and 109, organize test cases by their SARD ID number. Each test case has its own subdirectory. The subdirectories are organized by the millions digits, then thousands digits, then units digits. For example, the path to the first test case is `000/145/277/`, and the path to the second is `000/240/543/`.

Many of the cases were not changed from Version 1.2. If the case was not changed, the Version 1.2 case is used and the SARD ID number remains. If the case was changed, we *deprecated* the 1.2 case in the SARD and added a new case to the SARD.

Each test case has a unique file name. The file name consists of the CWE number and name, two underscores (`__`), followed by various identifying types, functions, and alternatives, then a control flow variant number. Control flow variant numbers are the same across the entire Juliet suite. For instance, `_03` variants wrap the target code in a conditional: `if (5 == 5)`.

Most test cases consist of a single file, but some span multiple files. Those with multiple files use a one-letter suffix. For instance, 77913 consists of four files:
`CWE127_Buffer_Underread__malloc_char_memmove_53a.c`,
`CWE127_Buffer_Underread__malloc_char_memmove_53b.c`,
`CWE127_Buffer_Underread__malloc_char_memmove_53c.c`, and
`CWE127_Buffer_Underread__malloc_char_memmove_53d.c`.
Instead of a one-letter suffix, some Java test cases use other suffixes, e.g., `_bad`, `_base`, or `_goodG2B`.

Additional information can be found in Boland and Black [9].

Each test case has a particular structure. Each has a single function intended to manifest a bug and has one or more functions with similar behavior, but with no bug. In this document we refer to the buggy code as *bad* code and the bug-free code as *good* code.

Problems were reported in both bad code and good code. Some code in Version 1.2 did not have the intended bug, or it had unintentional bugs. Typically, we show a bit of the code from Version 1.2, which we refer to as *old* code, and the corresponding bit from Version 1.3, which we refer to as *new* code.

1.2 Summary of Changes and Remaining Issues

This section summarizes the changes made to Juliet Version 1.2 to create Version 1.3. Section 2 details each change. This section also summarizes the problems that we know are still in Version 1.3. Section 3 details the problems and issues.

Version 1.2 had no test cases of integer overflow using unary increment (`i++` and `++i`) operators or test cases of integer underflow using decrement (`i--` and `--i`) operators. We created 3404 Java test cases (5612 files) and 2736 C test cases (4032 files) to manifest overflow or underflow. We added overflow cases under CWE190 and underflow cases under CWE191. For details, see Sec. 2.1.

• Fixed 104 C cases to actually have buffer overflow (BOF) [10, 11]. Also fixed CWE-121 Stack-based Buffer Overflow cases to allocate on the stack (Sec. 2.2).

• Fixed 144 C cases that had unintended BOF/Read/Above from constant strings in 64-bit architectures (Sec. 2.3).

• Added a simple check for allocation failure (NULL pointer) to 11 619 C files across 20 CWEs (Sec. 2.4).

• Removed 24 C BOF/Stack cases (51 files) under CWE121 that allocated memory on the stack in a subfunction, then used it after its lifetime–after return. We could not identify a way to fix the cases and still fulfill their test purposes (Sec. 2.5).

• Fixed 168 C files to not access memory after its lifetime (Sec. 2.6).

• Fixed 294 C files to initialize *both* members of a structure (Sec. 2.7).

• Fixed the C random number macros so their behavior was well defined (Sec. 2.8).

• Fixed 5200 C test cases (8120 files) under CWE078 to have OS injection on Unix (Sec. 2.9).

• Improved 72 C files to use `mkstemp()` as a more secure way to create temporary files (Sec. 2.10).

• Fixed 610 C files in 576 cases to correctly guard against possible overflow. Because the problems and fixes differ for different types, we detail changes for `int` type cases (Sec. 2.11), for `int64_t` cases (Sec. 2.12), and for `unsigned int` cases (Sec. 2.13) separately.

• Fixed 672 files to use `swprintf` instead of `snprintf()` to handle wide character string formats (Sec. 2.14).

The following changes did not invalidate the test cases, that is, not serve as a test for the intended bug, or add unintentional serious bugs. However, they improved Juliet and were worth making.

• Fixed code to use the correct format specifier in `fscanf()` for variables of type `int64_t` (352 files) and `size_t` (200 files). Also fixed the utility file `io.c` to use the correct format specifiers for those types. In addition, changed `io.c` to include files to properly declare macros (Sec. 2.15).

• Changed the compile ("make") process to be far more efficient. Also made other improvements and clean-ups (Sec. 2.16).

The astute reader may wonder why there were so many changes to C cases and not many to Java cases. We propose several reasons. First, memory allocation in C is very prone to errors, and most such errors cannot occur in Java. Second, C is an older language with many nuances about format and types that caused problems. Third, some problems that could have been present in the Java cases, such as wrong check for value out of range (Sec. 2.11), were not present; the code was correct in the earliest version.

Juliet Version 1.3 contains numerous changes from Version 1.2. However, many issues remain in Version 1.3. These are detailed in Sec. 3. Here is a summary of each issue. Thousands of cases have minor memory leaks (Sec. 3.1). Many cases have faults regardless of the input or do not exhibit failure at all (Sec. 3.2). Hundreds of cases access memory after its lifetime (Sec. 3.3) or have out-of-range checks that are still wrong (Sec. 3.4 and Sec. 3.5). A few cases have the wrong format specifier for wide strings (Sec. 3.6) or unintentional dead stores (Sec. 3.10). A utility function incorrectly prints the value 255 (Sec. 3.7). Many intentional bugs are removed in the good code by using a hardcoded value (Sec. 3.8). Some Java cases potentially leak stream resources (Sec. 3.9). There is no metadata indicating thousands of instances of dead code (Sec. 3.11) or hundreds of intentional integer overflows (Sec. 3.12). Temporary file names are still not fully secure (Sec. 3.13).

2. Details of All Changes

This section details each change to Version 1.2. The amount and kind of comment or description differs for each problem. For instance, some problems include an exhausting explanation of exactly why something is a bug. Others include how we gained assurance that all instances of a mistake were fixed or that there were no unintentional changes.

We provide the number of test cases or files associated with each change, usually listing them for future review. When the names of the files follow a certain pattern, we give the pattern using shell file name completion "star" (*) notation.

We usually edit the code that we include for examples to make it fit the printed page and to eliminate superfluous parts, so the reader may grasp the essentials more easily. Complete code is always accessible from the SARD.

Several people pointed out problems or made suggestions over the years following the release of Version 1.2. For attribution and historical purposes, most changes include a few words on the source and a convenient designation. Pascal Cuoq designated his comments with letters. André Maroneze used numbers. Eric Trapnell collected many comments and suggestions, and we tracked some of our work by row number in his spreadsheet.

2.1 Add Prefix and Postfix Increment Overflow and Decrement Underflow Cases

While tracking down a bug in a project that used Juliet, we realized that there were no test cases of integer overflow involving a prefix increment (`++i`) or postfix increment (`i++`) operator or of underflow involving a prefix decrement (`--i`) or postfix decrement (`i--`) operator.

There were cases for overflow and underflow for other operations, such as multiplication or addition:

```
int result = data + 1;
```

For C, we created prefix increment overflow cases from CWE190 `*_add_*.c` cases, named them `*_preinc_*.c`, and placed them under CWE190 in a new subdirectory, s06. Prefix increment cases were a straight-forward syntactic substitution. The postfix increment cases were more subtle since the variable value changes after the value is retrieved. One alternative is to use the comma operator to put everything on one line:

```
int result = (data++, data);
```

However, we thought this construct was too unusual, so we added another line with the actual increment:

```
data++;
int result = data;
```

We created postfix increment overflow cases from the prefix increment cases, and named them `*_postinc_*.c`. We placed them in another new subdirectory, s07.

The cases of underflow from the prefix and postfix decrement operators were analogous. We began with CWE191 `*_sub_*.c` cases and placed the new cases under CWE191, `_postdec_` in s04 and `_predec_` in s05.

Java cases came from CWE190 `*_add_*.java` and CWE191 `*_sub_*.java`. We placed them under CWE190, `_postinc_` in s06 and `_preinc_` in s07, and CWE191, `_postdec_` in s04 and `_predec_` in s05.

This added 684 test cases (1008 files) in each new C subdirectory and 851 cases (1403 files) in each new Java subdirectory, for a total of 2736 C cases (4032 files) and 3404 Java cases (5612 files).

2.2 Missing BOF

We use attributes of the Bugs Framework [10] Buffer Overflow (BOF) [11] class to classify bugs that are variously referred to as buffer overflow, out-of-bounds read, incorrect access of indexable resource, etc. These attributes are orthogonal and include access (either read or write), boundary (below/before or above/after), and location (heap, stack, etc.). Hence the title of this section may be read as "Missing Buffer Overflow."

There are 96 C cases (104 files) under CWE122 s06 and CWE121 s01 that test the misuse of `strlen()` with wide character strings, which is CWE-135. As written, they did not cause buffer overflow. The following example of old code is from `CWE122_Heap_Based_Buffer_Overflow__CWE135_01.c`, which is deprecated 70400. (We note "deprecated" since it may not appear in default SARD searches.)

```
size_t dataLen = strlen((char *)data);
void * dest = (void *)calloc(dataLen+1, 1);
memcpy(dest, data, (dataLen+1));
```

It has several problems. Note that `data` is a wide string. That was the intended bug; `strlen()` stops too early on wide strings and gives an incorrectly short length. `dest` was intended to be too small because the length is short. It is also too small because only one byte per character is allocated. However, using `memcpy()` prevents any buffer overflow! It copies the same number of bytes that were allocated.

The following example of new code is from `CWE122_Heap_Based_Buffer_Overflow__CWE135_01.c` 232119:

```
size_t dataLen = strlen((char *)data);
void * dest = (void *)calloc(dataLen+1, sizeof(wchar_t));
(void)wcscpy(dest, data);
```

In the new code, `calloc()` allocates wide characters (but still not enough). More importantly, `wcscpy()` copies the whole wide string, which is BOF/write [11].

While working on this problem, we noticed that CWE-121 is *Stack*-based buffer overflow, but the buffer is allocated in the heap. We changed those cases to use `alloca()`, via a macro, instead of `calloc()`. The following example is from `CWE121_Stack_Based_Buffer_Overflow__CWE135_01.c` 231402.

```
void *dest = (void *)ALLOCA((dataLen+1) * sizeof(wchar_t));
```

To correct a different problem, we also added code to check if the allocation succeeds. See Sec. 2.4 for details.

All of the test cases had `__CWE135_` in their names. We deprecated and replaced SARD IDs 62948 to 62995 and 70400 to 70447.

Pascal Cuoq reported this problem on 25 June 2013 (comment C), Takashi Matsuoka reported it on 22 August 2013, and André Maroneze reported it on 12 June 2017 (suggestion 7). This problem was Eric Trapnell's rows 2 and 12.

2.3 Unintended BOF/Read/Above for 64-bit Architectures

In 64-bit architectures, 144 C cases have unintended BOF/Read/Above [11] from constant strings. The following old code is from `CWE121_Stack_Based_Buffer_Overflow__char_type_overrun_memcpy_01.c` deprecated 63036:

```
#define SRC_STR "0123456789abcde0123"

typedef struct _charVoid
{
    char charFirst[16];
    void * voidSecond;
    void * voidThird;
} charVoid;

{
    charVoid structCharVoid;
```

To test that a static analysis tool recognizes the basic difference between a vulnerability and no vulnerability, the use of hardcoded "inputs" is reasonable acceptable. But eliminating the problem using a hardcoded value changes the program behavior (drastically!). These "fixes" are very different from the patches one would find in real code.

Damien Cupif points out that these pseudo-fixes tend to invalidate discrimination calculations. That is, the behavior of the good version is significantly different than the bad version behavior, not just for buggy values.

Expanding the utility of the Juliet suite may require rethinking the tactic of eradicating problems with hardcoded values. Finding a good resolution would require extensive consideration and changing thousands of files in a dozen different ways.

3.9 Uncaught Java Exceptions

We know of at least two uncaught Java exceptions in Juliet 1.3. In the following example, the constructor `OutputStreamWriter` can throw an exception that is not caught, creating a potential resource leak. The following example code is from `CWE400_Resource_Exhaustion__getParameter_Servlet_write_72b.java` 138404:

```
File file = new File("badSink.txt");
OutputStreamWriter writerOutputStream = new
           OutputStreamWriter(streamFileOutput, "UTF-8");
```

If the Java installation does not support `UTF-8`, the method exits, but the file remains open. This problem was Eric Trapnell's row 3.

Similarly, the constructor `InputStreamReader` can throw an exception. The resource is a URL connection in this example from `CWE400_Resource_Exhaustion__URLConnection_for_loop_14.java` 139105:

```
URLConnection urlConnection =
     (new URL("http://www.example.org/")).openConnection();

readerInputStream = new
     InputStreamReader(urlConnection.getInputStream(),
                                                  "UTF-8");
```

These are unlikely to cause problems in any execution of these test cases.

Aurélien Delaitre reported both of these problems in connection with SATE V [17]. This problem was Eric Trapnell's row 4.

3.10 Dead Stores

Forty cases have unintentional dead stores. That is, a value is stored in a variable, and the value is overwritten before it is used. There is no way to fix these and still keep the code similar to other variants. For example, here is bad code that does not use a variable. It is from `CWE563_Unused_Variable__unused_uninit_variable_char_33.cpp` 105689:

```
char data = dataRef;
/* FLAW: Do not use the variable */
```

The good code initializes then prints `data`.

```
char data = dataRef;
/* FIX: Initialize then use data */
data = 'C';
printHexCharLine(data);
```

This occurs for two scenarios (`unused_uninit_variable` and `unused_value`), four types (`int`, `char`, `wchar_t`, and `long`), and four code variants (33, 72b, 73b, and 74b). The corresponding files are named `CWE563_Unused_Variable__SCENARIO_TYPE_VARIANT.cpp`.

We could not think of a way to fix the dead store so that they are still similar to the 01 variants and also have the same structure as the thousands of other 33 variants. One approach is just to remove these tests: there are many other test cases with unused variables, unused values, dead stores, etc. Another approach is to not declare the local variable and use the right hand side of the assignment instead. So the above good code would become:

```
/* FIX: Initialize then use data */
dataRef = 'C';
printHexCharLine(dataRef);
```

We decided to leave the dead stores and note them as extraneous weaknesses.

Aurélien Delaitre reported this problem on 6 August 2015. This problem was Eric Trapnell's row 18.

3.11 Dead Code not in Metadata

There are thousands of cases with dead code, but the metadata that accompanies test cases and test suites does not note it. The following example code is from `CWE190_Integer_Overflow__int_File_postinc_02.java` 249242:

```
if (false)
{
    /* INCIDENTAL: CWE 561 Dead Code, the code below will
     * never run but ensure data is inititialized ... */
    data = 0;
}
else
{
    /* FIX: Use a hardcoded number that won't cause
       underflow, overflow, divide by zero, or loss-of-
       precision issues */
    data = 2;
}
```

For consistency and completeness, such dead code should be noted in the metadata for automated checking.

3.12 Integer Overflow not in Metadata

CWE-680 is a chain of two faults, an integer overflow (FRS/Overflow) that leads to a BOF/Write. All cases under CWE680 have metadata for BOF/Write at the right line. However, the FRS/Overflow is *not* in the metadata. The following example code is from `CWE680_Integer_Overflow_to_Buffer_Overflow__malloc_rand_01.c` 241054:

```
intPointer = (int*)malloc(data * sizeof(int));
for (i = 0; i < (size_t)data; i++)
{
    intPointer[i] = 0;
}
```

The computation for the amount of memory to allocate may overflow, causing a buffer that is too small to be allocated.

3.13 Temporary Files Still Not Secure

In Sec. 2.10, we explained how temporary files in Juliet 1.3 are more secure than those in Juliet 1.2. There we also explained the two remaining problems. We mention them again here to list all known problems in a single section.

First, we did not change any `wchar_t` cases because there is no widely-used equivalent of `mkstemp()` that handles `wchar_t` names.

Second, `mkstemp()` is not entirely secure. The file names are too predictable.

3.14 Suggestions We Did Not Take

In addition to comments and known problems, we received a number of suggestions on which we did not take action. This section records them.

3.14.1 Add `RAND16` and `RAND8` Macros

In 402 files of Juliet Version 1.3, `data` is assigned a random `char` value. In other files, `data` is assigned a random `short` value. The following example code is from `CWE190_Integer_Overflow__char_rand_postinc_01.c` 235991:

```
data = (char)RAND32();
```

The cast narrows the integer returned by `RAND32()` to a `char`, which changes some values. The C11 standard states that the exact nature of the changes is left to the implementation. In suggestion 1, André Maroneze said that users frequently request that such cases of implementation-dependent behavior be reported. To make it clear that there is no problem intended in these cases, he suggested adding new macros, for example:

```
#define URAND15() (rand() \% (1<<15))
#define RAND16() (rand() & 1 ? (short)URAND15() :
                            -((short)URAND15()) - 1)
```

```
#define URAND7() (rand() \% (1<<7))
#define RAND8() (rand() & 1 ? (char)URAND7() :
                              -((char)URAND7()) - 1)
```

We improved existing macros for random numbers (Sec. 2.8). However, we did not add specific macros to produce random `char` or `short` values.

3.14.2 Move Cases of NULL Check After Dereference

As explained in Sec. 2.4, 18 cases intentionally check a pointer for NULL *after* the pointer is dereferenced. In suggestion 7, André Maroneze suggested moving these cases from `CWE476_NULL_Pointer_Dereference` to `CWE571_Expression_Always_True`. We decided that CWE-571 is not a sufficiently close match. In fact, there is no CWE for this problem.

This suggestion was Eric Trapnell's row 15.

3.14.3 Move Cases of Incorrectly Calculating Multi-Byte String Length

André Maroneze's suggestion 7 was to move cases of incorrectly calculating multi-byte string length to a new directory named for CWE-135 Incorrect Calculation of Multi-Byte String Length. The following example of Juliet 1.3 code is from `CWE122_Heap_Based_Buffer_Overflow__CWE135_01.c` 232119:

```
size_t dataLen = strlen((char *)data);
void * dest = (void *)calloc(dataLen+1, sizeof(wchar_t));
(void)wcscpy(dest, data);
```

This affects 172 files in 96 cases, all with `__CWE135_` in their names. They are all under CWE121 and CWE122.

The original catalyst for looking at them was that the cases did not have BOF [11]. This problem was fixed so that the cases have BOF (Sec. 2.2).

The misuse of `strlen()` on wide character strings is intentional. We decided not to create a new directory and move these cases there for several reasons. First, these are legitimate BOF cases, so there is reason to leave them where they are or to duplicate them. Second, if we did create a new directory, it should contain a thorough set of cases of incorrectly calculating, not just one example. For instance, perhaps the misuse should be in other contexts, like printing and reading, not just copying. Third, for consistency, we would rename all of the files to start with `CWE135_*`) and change the names of the functions in the code. Finally, we plan to eventually replace the CWE classification with BF classification.

4. Some Thoughts on the Future of Juliet and Test Suites

In this section, we provide some thoughts on the future of the Juliet test suite and assurance tool testing in general.

There is no plan to create a Juliet Version 1.4. On one hand, many known problems can be corrected with techniques used to create Version 1.3. On the other hand, the Center for Assured Software plans to generate future test suites on demand. That is, a custom set of tests will be generated for each user. Custom sets of tests reduce the incentive to code a tool to an unchanging test suite.

With oversight and direction by NIST, students at TELECOM Nancy, a computer engineering school of the Université de Lorraine, Nancy, France implemented and then improved a test case generator [7]. They used the generator to create suites similar to Juliet for PHP and C# [18].

What would be the ultimate test suite? As Cohen et. al. explained [17], a perfect collection has three aspects: it represents production software, we know where all the bugs are, and it has lots of different types of bugs in varied situations. Juliet incorporates the last two aspects. Software that is used in production is typically large and complex. Juliet cases are *far* smaller and less complex than production software. Most synthetic or generated collections will be similarly small and less complex.

An approach to achieving all three aspects is to inject bugs into production software. Automated tools can help by finding locations with desirable execution flow, program state, and data visibility. However, none of the published approaches appear to have a path to completely automated bug injection of many types of bugs.

Acknowledgments

We thank all those who reported problems in Juliet 1.2, especially Pascal Cuoq (Pascal.CUOQ@cea.fr), Takashi Matsuoka (takashi.matsuoka@redlizards.com), André Maroneze (Andre.OLIVEIRAMARONEZE@cea.fr), Bertrand Stivalet, Aurélien Delaitre, and Elisa Heymann (elisa@cs.wisc.edu). We also thank Eric Trapnell and Charles D. De Oliveira for their work on Juliet 1.3.

References

[1] Center for Assured Software (2012) *Juliet Test Suite for C/C++ - Changelog*. Accessed 19 March 2018. URL https://samate.nist.gov/SARD/view.php?tsID=86.

[2] Common weakness enumeration: A community-developed list of software weakness types. Accessed 4 January 2018. URL https://cwe.mitre.org/.

[3] Center for Assured Software (2012) *Juliet Test Suite for Java - Changelog*. Accessed 19 March 2018. URL https://samate.nist.gov/SARD/view.php?tsID=87.

[4] Center for Assured Software (2012) *Juliet Test Suite v1.2 for Java User Guide*. Accessed 19 March 2018. Also available with the Juliet Test Suite v1.3 for Java. URL https://samate.nist.gov/SARD/resources/Juliet_Test_Suite_v1.2_for_Java_-_User_Guide.pdf.

[5] Center for Assured Software (2012) *Juliet Test Suite v1.2 for C/C++ User Guide*. Accessed 19 March 2018. Also available with the Juliet Test Suite v1.3 for C/C++. URL https://samate.nist.gov/SARD/resources/Juliet_Test_Suite_v1.2_for_C_Cpp_-_User_Guide.pdf.

[6] Software assurance reference dataset (SARD). Accessed 4 January 2018. URL https://samate.nist.gov/SARD/.

[7] Black PE (2017) SARD: Thousands of reference programs for software assurance. *Journal of Cyber Security and Information Systems - Tools & Testing Techniques for Assured Software - DoD Software Assurance Community of Practice: Volume 2* 5(3):6–13.

[8] Bojanova I, Black PE, Yesha Y (2017) Cryptography classes in bugs framework (BF): Encryption bugs (ENC), verification bugs (VRF), and key management bugs (KMN). *2017 IEEE 28th Annual Software Technology Conference (STC)*, pp 1–8. https://doi.org/10.1109/STC.2017.8234453. Gaithersburg, Maryland

[9] Boland T, Black PE (2012) Juliet 1.1 C/C++ and Java test suite. *IEEE Computer* 45(10):88–90.

[10] Bojanova I, Black PE, Yesha Y, Wu Y (2016) The bugs framework (BF): A structured approach to express bugs. *2016 IEEE International Conference on Software Quality, Reliability, and Security (QRS)*, pp 175–182. https://doi.org/10.1109/QRS.2016.29. Vienna, Austria

[11] Buffer overflow (BOF) class. Accessed 20 March 2018. URL https://samate.nist.gov/BF/Classes/BOF.html.

[12] Black PE, Ribeiro A (2016) SATE V Ockham sound analysis criteria. National Institute of Standards and Technology, NIST IR 8113. https://doi.org/10.6028/NIST.IR.8113. Noted 23 June 2017.

[13] Injection (INJ) class. Accessed 22 March 2018. URL https://samate.nist.gov/BF/Classes/INJ.html.

[14] dash — command interpreter (shell). Accessed 22 January 2018. URL http://manpages.ubuntu.com/manpages/xenial/en/man1/sh.1.html.

[15] IEEE (2017) *1003.1-2017 Portable Operating System Interface (POSIX)*.

[16] (2011) ISO/IEC 9899:2011 programming languages - C, Committee Draft — April 12, 2011 N1570. The International Organization for Standardization and the International Electrotechnical Commission (ISO/IEC) Joint Technical Committee JTC 1, Information technology, Subcommittee SC 22, Programming languages, their environments and system software interfaces, Working Group WG 14 - C.

[17] Cohen TS, et al. (2017) Improving software assurance through static analysis tool expositions. *Journal of Cyber Security and Information Systems - Tools & Testing Techniques for Assured Software - DoD Software Assurance Community of Practice: Volume 2* 5(3):14–22.

[18] Stivalet B, Fong E (2016) Large scale generation of complex and faulty PHP test cases. *Proc. 2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pp 409–415. https://doi.org/10.1109/ICST.2016.43