NIST Technical Note 1995

Juliet 1.3 Test Suite: Changes From 1.2

Paul E. Black

NIST

**National Institute of
Standards and Technology**

U.S. Department of Commerce

NIST Technical Note 1995

Juliet 1.3 Test Suite: Changes From 1.2

Paul E. Black
Software Quality Group
Software and Systems Division
Information Technology Laboratory

June 2018
INCLUDES UPDATES AS OF 08-22-2019; SEE PAGE iii

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.

Table of Contents

Change Log

August 2019

Corrected the subdirectories where we placed the new prefix increment and decrement cases. Section 2.1.

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 Version 1.1.

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).

4

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 available 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.

5

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, s07. 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, s06.

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;
```

7

```
memcpy ( structCharVoid . charFirst ,
                        SRC_STR , sizeof ( structCharVoid ));
```

The string `SRC_STR` is 20 characters long, including the NULL. A 64-bit architecture could have 64-bit (8 byte) pointers. `charVoid` is then $16 + 8 + 8 = 32$ bytes, so the old code read 32 characters from a 20 character string. To fix this, we extended the string to 32 characters:

```
#define SRC_STR "0123456789 abcdef 0123456789 abcde"
```

There are 36 cases under each of CWE121 s01 and CWE122 s01 (18 `char_*_memcpy` cases and 18 `char_*_memmove` cases respectively), and 36 under each of CWE121 s09 and CWE122 s09 (also 18 `char_*_memcpy` cases and 18 `char_*_memmove` cases). We deprecated and replaced SARD cases 63036 to 63071 and 67448 to 67483.

Pascal Cuoq explained this problem in his email of 19 March 2014. This problem was Eric Trapnell's rows 5 and 6.

2.4 Check for Allocation Failure

Thousands of C cases allocated memory. Few checked whether the allocation succeeded or failed. They just used the memory. Here is an example from `CWE401_Memory_Leak__struct _twoIntsStruct_malloc_01.c` deprecated 100474:

```
data = ( struct _twoIntsStruct *)
            malloc (100* sizeof ( struct _twoIntsStruct ));
data [0] . intOne = 0;
```

If the allocation fails, a NULL pointer is returned. Dereferencing a NULL pointer causes an undefined state. In the C language, "undefined" is more drastic than "the result may be any number." It means that following the dereference of a NULL pointer, "the program can whistle 'Happy Birthday' in all the colors of the rainbow and still be considered to conform to the standard." [12]

In particular, Frama-C halted analysis at the second statement above. After that, *anything* is allowed, so no precise analysis is reasonable. For the Static Analysis Tool Exposition (SATE) V Ockham Sound Analysis Criteria [12], Frama-C was run twice: once with allocation modeled as always succeeding, and once with allocation modeled as failing. The union of those two runs served the intended purposes.

We added simple checks after `malloc()`, `calloc()`, or `realloc()`. The check was minimal, as shown in the new version, 239815, which replaced the former case:

```
data = ( struct _twoIntsStruct *)
            malloc (100* sizeof ( struct _twoIntsStruct ));
if ( data == NULL ) { exit ( -1);}
data [0] . intOne = 0;
```

This check enables sound tools to continue analysis, since the state is always well defined.

We changed 11 619 files under 20 CWEs. We do not detail the files changed. Table 3 summarizes the number of files and test cases changed under each CWE.

Table 3. Number of files and test cases under each CWE directory to which we added checks for allocation failure.

Number of files	Number of cases	CWE subdirectory name
4	4	`CWE121_Stack_Based_Buffer_Overflow`
3342	3186	`CWE122_Heap_Based_Buffer_Overflow`
500	480	`CWE124_Buffer_Underwrite`
300	288	`CWE126_Buffer_Overread`
500	480	`CWE127_Buffer_Underread`
312	288	`CWE194_Unexpected_Sign_Extension`
312	288	`CWE195_Signed_to_Unsigned_Conversion_Error`
72	72	`CWE244_Heap_Inspection`
18	18	`CWE364_Signal_Handler_Race_Condition`
1008	972	`CWE401_Memory_Leak`
312	288	`CWE415_Double_Free`
150	150	`CWE416_Use_After_Free`
251	242	`CWE457_Use_of_Uninitialized_Variable`
54	54	`CWE467_Use_of_sizeof_on_Pointer_Type`
18	18	`CWE479_Signal_Handler_Use_of_Non_Reentrant_Function`
312	288	`CWE680_Integer_Overflow_to_Buffer_Overflow`
234	234	`CWE758_Undefined_Behavior`
576	576	`CWE761_Free_Pointer_Not_at_Start_of_Buffer`
2784	2694	`CWE762_Mismatched_Memory_Management_Routines`
560	480	`CWE789_Uncontrolled_Mem_Alloc`

Some test cases intentionally have possible NULL pointer dereference. We did not change them.

Eighteen cases under CWE476 intentionally omit checks immediately after allocation in both bad and good code. These cases check whether an analyzer will warn about a NULL check *after* a dereference. If a pointer is NULL and is dereferenced, the program has trouble. Any NULL check after that is of little use. The following example code is from `CWE476_NULL_Pointer_Dereference__null_check_after_deref_01.c` 104778:

```
intPointer = (int *)malloc(sizeof(int));
*intPointer = 5;
if (intPointer != NULL)
{
    *intPointer = 10;
}
```

This situation may arise in code if at one time there was a check immediately after the allocation, but later maintenance added the dereference before the check. Another possibility is that the code is in a context where the reference is never NULL, i.e., it was checked much earlier. A NULL check following a dereference suggests sloppy code at best.

The bad code of many cases under CWE690 does not check for NULL, while the

good code checks for NULL. The following example code is from `CWE690_NULL_Deref_From_Return__fopen_01.c` 111122:

```
data = fopen("file.txt", "w+");
fclose(data);
```

If `fopen()` fails, it returns NULL.

As with the other changes, we used small scripts to modify the files. Since we changed so many files and there were so many different circumstances, we made several special checks to verify the edits. In particular, we did not want to add checks that would remove the intended bug and invalidate the test case. We manually reviewed dozens of files, whilst developing and testing the editing script. After editing, we checked that the number of allocations and the number of NULL checks matched. We carefully audited the mismatches. We also compiled every test case individually as a simple check of correctness.

Pascal Cuoq reported this problem on 22 July 2013 (comment M). André Maroneze reported it on 12 June 2017 (suggestion 4). This problem was Eric Trapnell's row 13.

A related problem is thousands of "memory leaks," i.e., memory is allocated, but never freed. We saw little benefit and great cost to fix this problem, so we did not deal with it. We provide details in Sec. 3.1.

2.5 BOF/Stack Accessing Memory After Its Lifetime

We removed 24 C cases (51 files) that allocated memory on the stack, then used the memory after its lifetime. The following example code is from `CWE121_Stack_Based_Buffer_Overflow__CWE131_memcpy_21.c` 62870:

```
static int * badSource(int * data)
{
    {
        data = (int *)ALLOCA(10);
    }
    return data;
}


void CWE121_Stack_Based_Buffer_Overflow__CWE131_memcpy_21_bad()
{
    . . .
    data = badSource(data);
    {
        int source[10] = {0};
        memcpy(data, source, 10*sizeof(int));
```

`ALLOCA()` is a macro for `alloca()`, which is not defined in C11 or POSIX. It allocates memory on the stack. In typical implementations, the end of `badSource()` will terminate the memory's lifetime.

These cases are under CWE121 s01. They are `CWE121*__CWE131_` functions `loop`, `memcpy`, and `memmove`, and variants 21, 22, 42, 43, 61, 62, 83, and 84. The purpose of these tests is to allocate memory in one function, then use the memory in another. We could not identify a way to fix these cases and fulfill their test purposes, so we removed them.

Bertrand Stivalet reported this problem on 10 March 2014. This problem was Eric Trapnell's row 8.

2.6 Memory Accessed After Its Lifetime

Cases with a total of 168 C files under CWE476 accessed automatically allocated memory after its lifetime. The following example good code is from `CWE476_NULL_Pointer_ Dereference__int64_t_01.c` deprecated 104652:

```
int64_t * data;
{
    int64_t tmpData = 5LL;
    data = &tmpData;
}
printLongLongLine(*data);
```

Memory for `tmpData` is automatically allocated (usually on the stack) when execution enters the block. Although the address is saved, the lifetime of the memory ends when execution leaves the block. Thus, the argument of the call to `printLongLongLine()` is an invalid dereference.

We fixed this by declaring the variable earlier, as shown in the new good code 240719:

```
int64_t * data;
int64_t tmpData = 5LL;
{
    data = &tmpData;
}
printLongLongLine(*data);
```

Pascal Cuoq reported this problem on 26 June 2013 (comment G). André Maroneze reported this problem on 12 June 2017 (suggestion 5). This problem was Eric Trapnell's row 10.

After Juliet 1.3 was released, André Maroneze reported another set of cases that accessed memory after its lifetime. We provide details in Sec. 3.3.

2.7 Uninitialized Structure Member

While tracking down what we thought were incorrect warnings of uninitialized variables for the SATE V Ockham Sound Analysis Criteria, we found a wide-spread problem. One type of structure has two members:

11

```
typedef struct _twoIntsStruct
{
    int intOne;
    int intTwo;
} twoIntsStruct;
```

In 1046 files, both fields are initialized:

```
for (i = 0; i < 100; i++)
{
    dataBuffer[i].intOne = 1;
    dataBuffer[i].intTwo = 1;
}
```

However, in 294 files, `intOne` is initialized twice, and `intTwo` is not initialized:

```
for (i = 0; i < 100; i++)
{
    source[i].intOne = 0;
    source[i].intOne = 0;
}
```

We believe this was unintentional. We changed 240 files under CWE121 s04 and s05 (the files are named `*__CWE805_*`) and under CWE476. We deprecated and replaced test cases 64792 to 65031.

Pascal Cuoq reported this problem on 26 June 2013 (comment H). André Maroneze reported this problem on 12 June 2017 (suggestion 6). This problem was Eric Trapnell's row 9.

2.8 Undefined Behavior in Random Number Macros

The behavior of macros that Juliet 1.2 used for random numbers, `RAND32` and `RAND64`, were undefined. The macros shift signed integers out of range. The following old code is from `testcasesupport/std_testcase.h`:

```
#define RAND32() ((rand()<<30) ^ (rand()<<15) ^ rand())
#define RAND64() ((rand()<<60) ^ (rand()<<45) ^ (rand()<<30)
                                            ^ (rand()<<15) ^ rand())
```

As explained in Sec. 2.4, Frama-C ceased analysis after this undefined behavior.

We replaced the macros following André Maroneze's suggestions. The shifts are done on unsigned integers so that the behavior is defined. A single call to `rand()` then picks whether to produce a positive or a negative number. The new code for `RAND32` is the following:

```
#define URAND31() (((unsigned)rand()<<30)
                              ^ ((unsigned)rand()<<15) ^ rand())
#define RAND32() ((int)(rand()&1 ? URAND31() : -URAND31() - 1))
```

Note that this does not guarantee numbers with very high entropy. Specifically, there are few guarantees for numbers that `rand()` produces. There are far better ways to generate random numbers, but this seems sufficient for Juliet. We tested both of the new macros with an extensive pseudo-random number generator test program, and they both passed.

Although we only changed one file, this corrected undefined behavior in 3440 C test cases under 16 CWE directories.

Java cases in Juliet use random booleans for conditionals, but do not use "large" random numbers.

André Maroneze reported this problem on 12 June 2017 (suggestion 1).

2.9   No Unix Command Injection

The INJ [13] test cases, under CWE078, did not work in Unix platforms. The code compiles and executes, but does not perform a command injection. The following example of the old code is from `CWE78_OS_Command_Injection__char_console_execl_41.c` deprecated 118447:

```
#define COMMAND_INT_PATH "/bin/sh"
#define COMMAND_ARG1 "ls"
#define COMMAND_ARG2 "-la"
#define COMMAND_ARG3 data

#define EXECL execl

    EXECL(COMMAND_INT_PATH, COMMAND_INT_PATH, COMMAND_ARG1,
                            COMMAND_ARG2, COMMAND_ARG3, NULL);
```

The program reads a string into `data` then executes the `EXECL()` statement. Suppose that the input is `*;date>GOTCHA`. The program executes this command:

```
/bin/sh ls -la *;date>GOTCHA
```

This is not valid: `sh` does not execute binaries. It "treats the first argument as the name of a file from which to read commands (a shell script)" [14].

Charles De Oliveira developed the following code, which works in Windows and in Unix. He changed the execution statement and two macro definitions. The following new code is from `CWE78_OS_Command_Injection__char_console_execl_41.c` 244499:

```
#define COMMAND_ARG1 "-c"
#define COMMAND_ARG2 "ls "
#define COMMAND_ARG3 data

    EXECL(COMMAND_INT_PATH, COMMAND_INT_PATH, COMMAND_ARG1,
                            COMMAND_ARG3, NULL);
```

The program reads a string, appends it to `ls` (in code not shown), and puts it into `data`. If the input is again `*;date>GOTCHA`, the program executes this command:

```
/bin/sh -c ls *;date >GOTCHA
```

After executing `ls`, this executes `date` and puts the output in a local file.

In the 520 files that use `popen()`, we also corrected the argument to `popen()`. The following example old code is from `CWE78_OS_Command_Injection__char_file_popen_01.c` deprecated 119479:

```
pipe = POPEN(data, "wb");
```

The argument `wb` is not POSIX standard [15].

The following new code is from 245515:

```
pipe = POPEN(data, "w");
```

We changed 5600 C files, 2120 C++ files, and 400 header (.h) files, totaling 8120 files, in 4800 test cases under CWE78. There were 840 files in each of eight subdirectories, s01 through s08, and 280 files in s09.

Elisa Heymann reported this problem on 1 October 2014. This problem was Eric Trapnell's row 7.

2.10 Insecure Temporary Files

Test cases under CWE377 intentionally insecurely create and open temporary files. The bad functions used `tempnam()`, `tmpnam()`, or `mktemp()`. The following bad code is from `CWE377_ Insecure_Temporary_File__char_mktemp_01.c` deprecated 97938:

```
char tmpl[] = "fnXXXXXX";

filename = MKTEMP(tmpl);

fileDesc = OPEN(filename, O_RDWR|O_CREAT,
                                    S_IREAD|S_IWRITE);
```

The old good functions were more secure than the old bad functions: they opened files with `O_EXCL`. The following old good code to open the file is from the same case as above:

```
fileDesc = OPEN(filename, O_RDWR|O_CREAT|O_EXCL,
                                    S_IREAD|S_IWRITE);
```

Using `mkstemp()` makes the code more secure still. (However, file names are still too predictable for this to be considered very secure, but it is better. We repeat this explanation in Sec. 3.13 for consistency.) The following new good code is from 239333:

```
char filename[] = "/tmp/fileXXXXXX";

int fileDesc = MKSTEMP(filename);
```

This change potentially alters how the cases can be used. The cases now use a more secure function; they do not just use a dangerous function more securely. This raises the possibility that these cases might serve, also or instead, for CWE-242 Use of Inherently

Dangerous Function. Currently the test cases under CWE242 only have variants of one function, `gets()`. We decided that since Juliet does not include some number of dangerous functions, there was no reason to take time using these cases there.

In checking for other possible uses, we found a problem in test cases under CWE459, which deals with incomplete cleanup after execution. The test cases do not remove a temporary file. The following example old bad code is from `CWE459_Incomplete_Cleanup__char_01.c` deprecated 104285:

```
char tmpl[] = "badXXXXXX";

filename = MKTEMP(tmpl);

pFile = FOPEN(filename, "w");
```

The good code is the same, except that the prefix of the file's name is "good" instead of "bad." To remove the unintentional bug, we changed these cases to use `mkstemp()`, too. The following new good code is from 240619:

```
char filename[] = "goodXXXXXX";

int fileDesc = MKSTEMP(filename);

pFile = FDOPEN(fileDesc, "w");
```

To compile in Windows, Eric Trapnell found an open source version of `mkstemp()`, which we added to each file.

We did not change any `wchar_t` cases. In those cases, the file name is declared as

```
wchar_t *filename;
```

No Unix version of `mkstemp()` handles `wchar_t`, and Windows does not have `mkstemp()`.

We changed 54 files under CWE377 to use `mkstemp()` in the good functions. They were named `CWE377_Insecure_Temporary_File__char_FUNC` with 18 files where `FUNC` is `tempnam`, 18 `tmpnam`, and 18 `mktemp`. We also changed 18 files under CWE459 to use `mkstemp()`. They were named `CWE459_Incomplete_Cleanup__char_01.c` to `_18.c`.

André Maroneze reported this problem on 12 June 2017 (suggestion 8). This problem was Eric Trapnell's row 11.

2.11 Wrong Check for Value Out of Range - `int`

Some of the good code under `CWE190_Integer_Overflow` intended to avoid overflow by checking that the value was in range. The following example of good code is from `CWE190_Integer_Overflow__int_fscanf_square_01.c` deprecated 83358:

```
if (abs((long)data) <= (long)sqrt((double)INT_MAX))
{
    int result = data * data;
}
```

15

```
else
{
    printLine("data value is too large ...
}
```

The problem is that the absolute value of the most negative integer (`abs(INT_MIN)`) is undefined because it is greater than `INT_MAX`. Some systems define `abs(INT_MIN)` to return `INT_MIN`, which is negative. In that case, it would pass the test anyway and cause an overflow. In addition, the function `abs()` does not handle `long` values; at best the cast is not needed. The following example of the check in new good code is from 235621:

```
if (data > INT_MIN &&
                    abs(data) < (long)sqrt((double)INT_MAX))
```

Variants of CWE190 cases comprise five operations: square (`data*data`), multiply (`data*2`), add (`data+1`), prefix increment (`++data`), and postfix increment (`data++`). Only the cases with square operation had this problem. All others used simpler checks that did not have a problem with `INT_MIN`.

The case variants comprise five types: `char`, `int64_t`, `int`, `short`, and `unsigned_int`. The types `char` and `short` are fine, in usual architectures, because `CHAR_MIN` and `SHORT_MIN` do have positive `int` representations. Unsigned integers are all positive and thus not a problem. The problem with `abs(min)` only occurs for types `int` and `int64_t`. Type `int64_t` variants had additional problems, so we discuss them in the next section, Sec. 2.12. Type `unsigned int` variants have a different problem, so we discuss them in Sec. 2.13.

In 30 test cases, the "input method" is just the hardcoded value `INT_MAX`. The following example old good code is from `CWE190_Integer_Overflow__int_max_square_01.c` deprecated 83646:

```
data = INT_MAX;
if (abs((long)data) <= (long)sqrt((double)INT_MAX))
{
    int result = data * data;
```

Eric Trapnell argued that because `data` can never be `INT_MIN` in these cases, the check need not be changed. Since these tests are for static analyzers, it may be informative to have as consistent code as possible, in case a known value of `INT_MAX` affects analysis. Accordingly, we changed those cases, too.

Using a hardcoded value to "fix" the good function may be a concern. This changes the functionality and is therefore not a fix as much as it is removing the bug. See Sec. 3.8 for more discussion about this.

For `int` type, we fixed 288 C files (in 288 test cases) under CWE190 under s02 (48 files), s03 (192 files), and s04 (48 files). A total of 228 files were in C, and 60 files were in C++. These were all named `CWE190_Integer_Overflow__int_*_square_*.c` or `.cpp`.

Pascal Cuoq reported this problem on 25 July 2013 (comment N). André Maroneze reported this problem on 12 June 2017 (suggestion 3). This problem was Eric Trapnell's row 14.

16

After Juliet 1.3 was released, Maroneze reported that many cases still had similar problems with range checks. We provide details in Sec. 3.5.

2.12   Wrong Check for Value Out of Range - `int64_t`

The old Juliet Version 1.2 code does not properly handle `int64_t` types, even with casts. The following example old good code is from `CWE190_Integer_Overflow__int64_t_fscanf_square_01.c` deprecated 82638:

```
fscanf (stdin, "%lld", &data);
if (abs((long)data) <= (long)sqrt((double)LLONG_MAX))
{
    int64_t result = data * data;
```

The functions `abs()` and `sqrt()` do not handle `int64_t` type values. The cast to `double` may distort values in other ways. The following new check is from 235405:

```
fscanf (stdin, "%" SCNd64, &data);
if (imaxabs((intmax_t)data) <= sqrtl(LLONG_MAX))
{
    int64_t result = data * data;
```

We also added an include of `inttypes.h` to all the files of these cases.

We also noticed that the `%lld` format specifier does not handle `int64_t` type values. We corrected it.

For `int64_t` type, we changed 178 C and C++ files (in 144 test cases) under CWE190 s02. Of those, 114 files were C, and 30 files were C++. These files were all named `CWE190_Integer_Overflow__int64_t_*_square_*.c` or `.cpp`.

The number of files fixed for `int64_t` type cases differs from the number fixed for the `int` type cases (Sec. 2.11). The `int` cases have input variants `connect_socket`, `fgets`, and `listen_socket`. The `int64_t` type cases do not have those input variants. Also, we changed `int64_t` files to fix a `scanf()` format specifier problem (Sec. 2.15).

Pascal Cuoq reported this problem on 25 July 2013 (comment P). This problem was Eric Trapnell's rows 14 and 16.

As we wrote this report, we realized that this fix does not solve the minimum integer problem, as explained above in Sec. 2.11. We note this as a problem remaining in Juliet 1.3 in Sec. 3.4.

After Juliet 1.3 was released, André Maroneze reported that many cases still had similar problems with range checks. We found 432 cases in Juliet 1.3 that we failed to fix. We provide details in Sec. 3.5.

As with the `int` cases, there are still a concern with some of these cases. One version of the good function is "fixed" by using a hardcoded value: `data = LLONG_MAX`. Sec. 3.8 discusses this in more detail.

2.13 Wrong Check for Value Out of Range - `unsigned int`

This check is, to paraphrase Pascal Cuoq, a theoretical problem that would only happen on an architecture with a 64-bit `int` type, which is rare. The following old checking code is from `CWE190_Integer_Overflow__unsigned_int_fscanf_square_01.c` deprecated 84366:

```
if (abs((long)data) <= (long)sqrt((double)UINT_MAX))
{
    unsigned int result = data * data;
```

When `UINT_MAX` ($2^{64} - 1$) is converted to `double`, it is rounded up to 2^{64}. The square root of 2^{64} is 2^{32}. If `data` is 2^{32}, it passes the check, and the multiplication overflows.

We followed Cuoq's recommendation and changed `<=` to `<`. The following example new code is from 235789:

```
if (abs((long)data) < (long)sqrt((double)UINT_MAX))
```

We changed 144 files under CWE190 s05, all named `__unsigned_int_*_square_`.

Pascal Cuoq reported this problem on 25 July 2013 (comment O). This problem was Eric Trapnell's row 15.

2.14 Wide Format Strings Mishandled

Some test cases pass wide character format strings to `snprintf()`, which does not handle them. The following old code is from `CWE122_Heap_Based_Buffer_Overflow__c_CWE805_wchar_t_snprintf_01.c` deprecated 72176:

```
#ifdef _WIN32
#define SNPRINTF _snwprintf
#else
#define SNPRINTF snprintf
#endif

    SNPRINTF(data, 100, L"%s", source);
```

We replaced `snprintf()` with `swprintf()` in 672 files. There are 188 files under CWE121 (in s05, s06, s07, and s08), 208 files under CWE122 (in s04, s05, s09, and s10), 240 files under CWE134 (in s03, s04, s05, and s06), 18 files under CWE252, and 18 files under CWE253. All of the cases have `_wchar_t_` in their names and have `_snprintf` just before the variation number.

Pascal Cuoq reported this problem on 26 June 2013 (comment K). This problem was Eric Trapnell's row 12.

As we wrote this report, we noticed that `source` itself is a wide string. It is handled with the wrong format specifier: `%s` instead of `%S`. This is a latent bug in Juliet 1.3, which we explain in Sec. 3.6.

18

2.15 Wrong `fscanf()` Format Specifier for `int64_t` and `size_t`

The `%lld` format specifier, used in `fscanf()`, does not handle `int64_t` type variables. The following example old code is from `CWE190_Integer_Overflow__int64_t_fscanf_square_01.c` deprecated 82638:

```
int64_t data;
fscanf (stdin, "%lld", &data);
```

We changed the code to use `SCNd64`. The following new input code is from 235405:

```
int64_t data;
fscanf (stdin, "%" SCNd64, &data);
```

We also added an include of `inttypes.h` in all the files of these cases. This occurs in 352 files in the following seven subdirectories: CWE190, subdirectories s01, s02, s06, and s07, and CWE191, subdirectories s01, s04, and s05. The files are named `CWE19[01]_Integer_{Over,Under}flow__int64_t_fscanf_FUNC_*`, where `FUNC` is add, sub, multiply, preinc, postinc, predec, or postdec.

Similar to that, the `%ud` format specifier is not the standard way to handle `size_t` variables. The following example of old code is from `CWE789_Uncontrolled_Mem_Alloc__new_char_fscanf_14.cpp` deprecated 117572:

```
size_t data;
    fscanf(stdin, "%ud", &data);
```

We replaced %ud with %zu. The following new code is from 243952:

```
size_t data;
    fscanf(stdin, "%zu", &data);
```

These are all under CWE789. The cases are named `CWE789_Uncontrolled_Mem_Alloc__{malloc,new}_{char,wchar_t}_fscanf_*.{c,cpp}`. The `__malloc_` cases are in s01. The `__new_` files are in s02. There are 100 files in each of s01 and s02.

André Maroneze reported this problem on 12 June 2017 (part of suggestion 2). This problem was Eric Trapnell's row 17.

Similar to those two problems, utility functions in the support file `io.c` used the wrong format specifiers. The following is the old code:

```
void printLongLongLine (int64_t longLongIntNumber)
{
    printf("%lld\n", longLongIntNumber);
}


void printSizeTLine (size_t sizeTNumber)
{
    printf("%ud\n", sizeTNumber);
}
```

19

We changed this code to use the appropriate format specifiers:

```
void printLongLongLine (int64_t longLongIntNumber)
{
    printf("%" PRId64 "\n", longLongIntNumber);
}

void printSizeTLine (size_t sizeTNumber)
{
    printf("%zu\n", sizeTNumber);
}
```

We also added an include of `inttypes.h`, to define `PRId64`, and `wctype.h`, to declare `iswxdigit()`.

Pascal Cuoq reported this problem as part of comment Q.

2.16 Improve Compile Files and Scripts

As explained in [5] and [4], Juliet was designed so that the user could either compile all test cases in one big executable (*All*) or each test case as its own, individual executable (*Ind*).

The Juliet 1.2 makefile command to make one big executable, *All*, for C/C++ on a Unix platform was impractical. The command passed more than 100 000 files to gcc, which gcc could not handle.

As one step to fix that, we changed the makefile command to generate individual test case executables, *Ind*, (using -DINCLUDEMAIN), in addition to one executable for each CWE. (One executable per CWE was already available in Juliet 1.2.) We changed this command to create unlinked object files (.o files) for each source file (.c or .cpp files). Following that all of the object files in a directory could be linked to create an executable of all of the test cases in that directory or all of the object files could be linked to create a `partial.o` file.

The `partial.o` in each directory is used by the new makefile command to link the one big executable, *All*. We heavily edited the command to link each directory's `partial.o`. This compilation process is far quicker than the process in Version 1.2 and consumes fewer resources.

Hence in the top directory, there are now two make commands:
```
$ make individuals
```
creates an executable file for each C/C++ test case.
```
$ make Juliet1.3
```
or
```
$ make
```
invokes `make individuals`, then creates a single executable with of all the test cases.

Similar options are supported for Windows in the .bat files, except that individual executables are not supported.

For the Java code, the most significant improvement is that there are now commands available in the top directory for compiling test cases:

```
$ ant compile
```

creates only bytecode files.

```
$ ant jar
```

creates java archive files.

```
$ ant war
```

creates web archive files.

We also made small changes to improve the manifest of jar files.

For both C and Java, we removed code that is not needed. Previously scripts in the top directory had code to ignore directories named svn. SVN is a source control system, like git or rcs. In SVN, a .svn subdirectory stores change metadata for each source code directory. Hence, .svn directories were scattered all over. Files in those directories should not be included in makefiles or other files. Since there are no .svn directories in Juliet 1.3, the code is unnecessary.

3. Known Problems in Juliet 1.3

We corrected many problems in the Juliet 1.2 test suite. This section details the many systematic problems remaining in Juliet 1.3 of which we know. We decided not to fix some because the effort to fix the code exceeded any benefit. Four problems were noticed only months after Juliet 1.3 was released. They are detailed in Sections 3.3, 3.4, 3.5, and 3.6.

3.1 Memory Leaks

The first problem is that thousands of cases have memory leaks. That is, memory is allocated, but never freed (until the program ends). The following example is from `CWE758_Undefined_Behavior__double_pointer_alloca_use_01.c` 112082:

```
static void good1()
{
    {
        double * data;
        double **pointer = (double **)ALLOCA(sizeof(double *));
        data = (double *)malloc(sizeof(double));
        *data = 5.0;
        *pointer = data;
        {
            double * data = *pointer;
            printDoubleLine(*data);
        }
    }
}
```

21

We decided not to fix this memory leak problem for several reasons. First, the memory leak does not lead to undefined behavior, and therefore should not stop analyzers from looking for other problems. Second, it should not cause any problem in practice. Even if all the cases are compiled and executed together, only about 2 megabytes of memory will be allocated.

Third and most importantly, it would take a *huge* amount of manual effort to write and test scripts to insert the proper `free()` commands at the right places. Worse, we foresee two mistakes possible with the scripts: the memory is freed too early or the wrong memory is freed. Either way, later code accesses freed memory. There is no simple way to check for these mistakes.

Aurélien Delaitre reported this problem on 15 September 2015. This problem was Eric Trapnell's row 19.

3.2 No Evident Failure

Although most cases have faults, i.e., corrupted internal states, many cases do not take user input or do not have externally apparent failures. The following bad code is from `CWE416_Use_After_Free__malloc_free_char_01.c` 240263:

```
data = (char *)malloc(100*sizeof(char));
. . . initialize . . .
free(data);
printLine(data);
```

Although this code always uses memory after it is freed, there is usually no visible failure, e.g., a crash or corrupted result. Even some cases that have apparent failures do so for *all* inputs. That is, they crash given essentially any input.

Future versions of these test cases might be built so that all of them execute reasonably for some inputs and fail for other inputs. Even better would be versions that have exploitable security vulnerabilities.

David Musliner reported this problem on 28 February 2018.

3.3 Memory Accessed After Its Lifetime

All 80 cases under CWE843 access automatically allocated memory after its lifetime. The following example code is from `CWE843_Type_Confusion__short_01.c` 122807:

```
void * data;
{
    short shortBuffer = 8;
    data = &shortBuffer;
}
printIntLine(*((int*)data));
```

Memory for `shortBuffer` is automatically allocated (usually on the stack) by the time that execution enters the block. Although the address is saved, the lifetime of the memory ends

22

when execution leaves the block. Thus, the argument to `printIntLine()` is an invalid dereference.

This can be fixed by declaring the variable earlier, as detailed in Sec. 2.6. This is how the fixed code might appear:

```
void * data;
short shortBuffer = 8;
{
    data = &shortBuffer;
}
printIntLine(*((int*)data));
```

André Maroneze reported this problem on 12 April 2018.

3.4 Check for Value Out of Range Still Wrong - `int64_t`

As we wrote this report, we realized that the range check fix explained in Sec. 2.12 does not solve the minimum integer problem for `int64_t`. In brief, the absolute value of `LLINT_MIN` is not defined by the C11 standard [16]. The check should be

```
if (data > LLINT_MIN &&
            imaxabs((intmax_t)data) <= sqrtl(LLONG_MAX))
```

This affects 144 files under CWE190 s02. All the files with this problem are named `CWE190*_int64_*square*`.

3.5 Additional Cases with Wrong Check for Value Out of Range

Some of the good code under `CWE190_Integer_Overflow` intended to avoid overflow by checking that the value was in range. The following example of good code is from `CWE190_Integer_Overflow__unsigned_int_max_square_01.c` 235831:

```
if (abs((long)data) <= (long)sqrt((double)UINT_MAX))
{
    unsigned int result = data * data;
    printUnsignedLine(result);
else
{
    printLine("data value is too large ...
}
```

The problem is in `abs((long)data)`. On some architectures, the largest unsigned integer will not fit in a `long`. Thus the cast does not behave as intended. In addition, `abs()` does not handle `long` values. The problem could be addressed by checking that the value fits and using an appropriate absolute value function, as explained in Sections 2.11 and 2.12. André Maroneze suggested the following code:

```
if (data <= LONG_MAX &&
            labs(data) < (long)sqrt((double)UINT_MAX))
```

The files with this problem are all under CWE190, subdirectories s01, s04, and s05. They are named `CWE190_Integer_Overflow__TYPE_SOURCE_square_VARIANT.c` or `.cpp`, with three types, `char`, `short`, and `unsigned_int`, three sources, `max`, `rand`, and `fscanf`, and 48 flow variants, for a total of 432 files.

André Maroneze reported this problem on 12 April 2018.

3.6 Wrong Format Specifier for Wide String

As we wrote this report, we noticed the change explained in Sec. 2.14 does not fix all the printing problems. The following code is from `CWE122_Heap_Based_Buffer_Overflow__c_CWE805_wchar_t_snprintf_01.c` 233407:

```
#define SNPRINTF swprintf

    wchar_t source[100];

    SNPRINTF(data, 100, L"%s", source);
```

The argument, `source`, is a wide string. The code has the wrong format specifier: `%s`. To handle wide strings, the specifier should be `%S`.

This problem needs to be fixed in 672 files under various directories: CWE121 (subdirectories s05, s06, s07, and s08), CWE122 (subdirectories s04, s05, s09, and s10), CWE134 (subdirectories s03, s04, s05, and s06), CWE252, and CWE253.

3.7 Wrong Format Specifier to Print `char` as Hexadecimal

In Juliet 1.3, utility code prints `char` type arguments with the wrong format specifier. The function `printHexCharLine()` is defined in the support utility file `io.c` as:

```
void printHexCharLine (char charHex)
{
    printf("%02x\n", charHex);
}
```

The format specifier `%x` is for an `unsigned int` argument. In this case, `charHex` is promoted to `int`. If type `char` is signed, a common choice that the C11 standard [16] leaves to each compiler, a negative value is promoted to a negative integer by sign extension. That means the above typically prints 255 as `ffffffff`.

Pascal Cuoq suggested using casts to print properly:

```
    printf("%02x\n", (unsigned int)(unsigned char)charHex);
```

The cast `(unsigned int)` is needed because the C11 standard 7.21.6.1:9 "The fprintf function" states, "If any argument is not the correct type for the corresponding conversion specification, the behavior is undefined."

As an alternative, C11 7.21.6.1:7 provides a length modifier, `hh`, to handle this:

```
    printf("%02hhx\n", charHex);
```

24

Pascal Cuoq reported this problem as part of comment Q. André Maroneze reported this problem on 12 June 2017 (part of suggestion 2).

3.8 Many Bugs Removed by Using Hardcoded Values

The most widespread and challenging issue in Juliet 1.3 is that the good code in thousands of cases removes a problem by just using a hardcoded value. For instance, some cases under CWE190 read a value then increment it. The following example bad code is from `CWE190_Integer_Overflow__int_fscanf_add_01.c` 83262:

```
fscanf(stdin, "%d", &data);
{
    int result = data + 1;
```

Note that there is no check for overflow. This particular case has two good functions, each with a different resolution. One adds a check:

```
fscanf(stdin, "%d", &data);
if (data < INT_MAX)
{
    int result = data + 1;
```

The other one merely sets the variable to a value that will not cause an overflow:

```
data = 2;
{
    int result = data + 1;
```

Hardcoded values to remove bugs are located throughout the Juliet test suite. For instance, cases under nineteen CWEs use

```
data = "foo";
```

We identified a dozen different kinds of hardcoded constants, such as integers, passwords, and strings.

Fig. 1. A cartoon highlighting that making a bug disappear is not the same as fixing it.
NON SEQUITUR © 2018 Wiley Ink, Inc.. Dist. By ANDREWS MCMEEL SYNDICATION. Reprinted with permission. All rights reserved.

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.

27

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

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