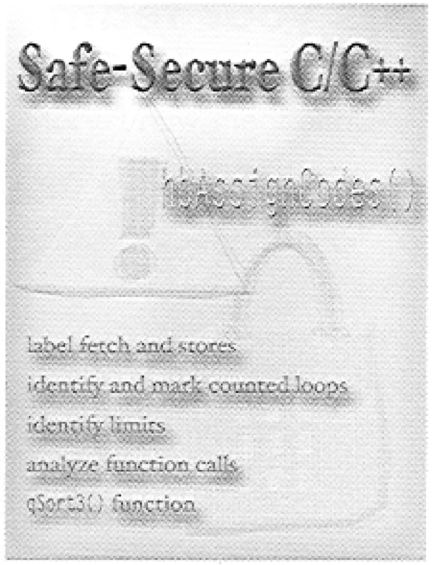

Robert C. Seacord

Validating C and C++ For Safety and Security

A structured approach to manual code review

Buffer overflows are a primary source of software vulnerabilities. A buffer overflow occurs when data is written outside of the boundaries of the memory allocated to a particular data structure. Buffer overflows are troublesome in that they can go undetected during the development and testing of software applications. Common C and C++ compilers neither identify possible buffer overflow conditions at compilation time nor report buffer overflow exceptions at runtime [1].

Not all buffer overflows lead to exploitable software vulnerabilities. However, a buffer overflow can cause a program to be vulnerable to attack when the program's input data is manipulated by a (potentially malicious) user. Even buffer overflows that are not obvious vulnerabilities can introduce risk.

Code inspections have been used for many years to reduce errors in program development [2]. Code inspections used primarily to identify and eliminate security flaws leading to exploitable buffer overflows and other vulnerabilities are referred to as "source-code security audits." These audits can be effective in finding and eliminating problems that cannot be detected using existing tools. However, source-code audits are typically unstructured and rely largely on the experience and tenacity of the programmers performing the review.

While any manual process is prone to error, following a more structured approach may produce a higher level of assurance that potential security flaws have been identified and properly remediated. In the remainder of this article, I describe a manual review process for C and C++ language programs that is based on Safe-Secure C/C++ from Plum Hall [3].

Robert C. Seacord is a senior vulnerability analyst at the CERT/C and author of *Secure Coding in C and C++* (Addison-Wesley, 2005). He can be reached at rcs@cert.org.

Safe-Secure C/C++ (SSCC) is a set of methods to eliminate vulnerabilities resulting from buffer overflows and other programming errors in C and C++ using a mixture of compile-time, link-time, and runtime tests, plus some design-time restrictions. The basic premise underlying SSCC is that most exploits (especially those that transfer control to arbitrary code) need to read or write to memory locations outside the bounds of the data structures defined by the program. This allows an attacker, for example, to overwrite the return address on the stack or other address to which control is eventually transferred to execute arbitrary code provided by the attacker or already resident on the system. By eliminating the possibility of such writes, it is possible to eliminate these vulnerabilities.

To demonstrate how the manual review process works, I apply it to the `hbAssignCodes()` function shown in Example 1 from the Standard Performance Evaluation Corporation (SPEC) C language benchmark program 256.bzip2.

To simplify the process and reduce the cognitive load for the reviewer, the manual review is carried out in a series of steps. Because SSCC is based on preventing reads and writes from outside the bounds of programmatically defined data structures, the first step is to identify fetch and stores that involve subscripting or dereferencing a pointer. The `hbAssignCodes()` function is shown in the right-hand side of Table 1.

There are two fetch and stores of interest in this function, both on line 1441. The variable `length` is a function parameter and is defined as a pointer to `unsigned char`. The variable `code` is also a function parameter but is defined as a pointer to `int`. When these variables are subscripted, the value of these pointers is added to `i` times the `sizeof` of the respective types. On 32-bit Intel Architecture (IA-32), for example, the `sizeof` of an `unsigned char` is a single byte, while an `int` is 4 bytes.

In both cases, there is no clear indication what these arguments point to, so these fetch and store operations could potentially be out of bounds. Consequently, both subscript operations are annotated in the left-hand side of the table. The `SUB4()` notation is shorthand for "is a suitable subscript for" and is both a requirement and a guarantee. This means that the developer, compiler, or runtime system must guarantee that this requirement is satisfied before line 1441 is executed, to eliminate the possibility of buffer overflow.

In step 2 we look for and mark counted loops. Counted loops are the most basic type of loop and involve a loop counter that monotonically increases or decreases until a maximum or minimum value is reached. If the loop counter is increasing, the loop is referred to as a "counted-plus loop." When the loop counter is decreasing, the loop is referred to as a "counted-minus loop." Table 2 identifies two occurrences of `counted-plus` loops in the `hbAssignCodes()` function.

Counted loops are interesting because they can help establish easy-to-identify limits. The variable `i` (used twice as an index on line 1441) is the loop counter for the `counted-plus` loop on line 1440. We see from the `for` statement that the value of `i` starts

(continued from page 6)

at 0 and increases monotonically until it is one less than the value of `alphaSize`. This means that the values up to `alphaSize-1` must be suitable as a subscript for both the `length` and `code` arrays. These limits are identified in the third step of the review process. Table 3 shows the annotated `hbAssignCodes()` function at the completion of step 3.

Table 1: *Step 1—Label fetch and stores.*

	1430	void hbAssignCodes (int *code,
	1431	unsigned char *length,
	1432	int minLen,
	1433	int maxLen,
	1434	int alphaSize)
	1435	{
	1436	int n, vec, i;
	1437	
	1438	vec = 0;
	1439	for (n = minLen; n <= maxLen; n++) {
	1440	for (i = 0; i < alphaSize; i++)
(a) i SUB4(length); (b) i SUB4(code)	1441	if (length[i] == n) { code[i] = vec; vec++; };
	1442	vec <<= 1;
	1443	}
	1444	}

Table 2: *Step 2—Identify and mark counted loops.*

	1430	void hbAssignCodes (int *code,
	1431	unsigned char *length,
	1432	int minLen,
	1433	int maxLen,
	1434	int alphaSize)
	1435	{
	1436	int n, vec, i;
	1438	vec = 0;
{ counted-plus }	1439	for (n = minLen; n <= maxLen; n++) {
{ counted-plus }	1440	for (i = 0; i < alphaSize; i++)
(a) i SUB4(length); (b) i SUB4(code)	1441	if (length[i] == n) { code[i] = vec; vec++; };
	1442	vec <<= 1;
	1443	}
	1444	}

Table 3: *Step 3—Identify limits.*

	1430	void hbAssignCodes (int *code,
	1431	unsigned char *length,
	1432	int minLen,
	1433	int maxLen,
	1434	int alphaSize)
	1435	{
	1436	int n, vec, i;
	1438	vec = 0;
{ counted-plus }	1439	for (n = minLen; n <= maxLen; n++) {
{ counted-plus }	1440	for (i = 0; i < alphaSize; i++)
(a) i SUB4(length), alphaSize SUB5(length); (b) i SUB4(code), alphaSize SUB5(code);	1441	if (length[i] == n) { code[i] = vec; vec++; };
	1442	vec <<= 1;
	1443	}
	1444	}

Table 4: *Step 5—Analyze function calls.*

	1012	unsigned char len [6][258];
		...
	1021	int code [6][258];
		...
{ nInUse is 256 so alphaSize = 258 }	1736	alphaSize = nInUse+2;
		...
{ counted-plus }	1898	for (t = 0; t < nGroups; t++) {
	1899	minLen = 32;
	1900	maxLen = 0;
{ counted-plus }	1901	for (i = 0; i < alphaSize; i++) {
nGroups SUB5(len); { alphaSize = 258 } alphaSize SUB5(len[]);	1902	if (*len[t][i]* > maxLen) maxLen = *len[t][i]*;
(a) [ok], (b) [ok]	1903	if (*len[t][i]* < minLen) minLen = *len[t][i]*;
	1904	} // end for i < alphaSize
{ alphaSize SUB5(len[]) is [ok] } { alphaSize SUB5(len[]) so alphaSize SUB5(code[]) is [ok] }	1907	hbAssignCodes (*&code[t][0]*, *&len[t][0]*,
	1908	minLen, maxLen, alphaSize);
	1909	} // end for t < nGroups

Table 3 adds annotations for line 1441 showing that `alphaSize` is "SUB5" for both the length and code arrays. "SUB5" means that the value is one greater than a value that is suitable as an array index (that is, "SUB4 plus one").

In step 4 we annotate the function's declaration to indicate whether there are any requirements on arguments to the function. Because the `alphaSize` argument must be SUB5 for both length and code, we need to annotate this requirement as shown in Example 2.

In step 5 we analyze each call to the function to determine whether the requirements imposed by the new annotations can be guaranteed. In the current example, there is only one call to this function on line 1907 of the program (as shown in Table 4 along with some other relevant lines from the sample program). The arguments to this function include the `code` and `len` arrays, respectively declared on lines 1012 and 1021, and `alphaSize`.

During step 3, it was also determined that `alphaSize` must be SUB5 for `len[]` (see the annotation for line 1902). Flow analysis shows that after line 1736, `alphaSize` equals 258, which provides the guarantee prior to invoking the `hbAssignCodes()` function on line 1868 that `alphaSize` is SUB5 for `len`. Because the `len` array has the same bounds as the `code` array, this also guarantees that `alphaSize` is SUB5 for

Example 1: hbAssignCodes() *function.*

```
void hbAssignCodes(
  int *code, unsigned char *length,
  int minLen, int maxLen, int alphaSize )  {
  int n, vec, i;
  vec = 0;
  for (n = minLen; n <= maxLen; n++) {
    for (i = 0; i < alphaSize; i++)
      if (length[i] == n) { code[i] = vec; vec++; };
        vec <<= 1;
  }
}
```

Example 2: *Step 4—Annotate the declaration.*

```
void hbAssignCodes(
  int *code, unsigned char *length,
  int minLen, int maxLen,
  int alphaSize /* SUB5(length) SUB5(code) */
);
```

Table 5: *The* qSort3() *function.*

	2433	void qSort3 (int loSt, int hiSt, int dSt)
	2434	{
	2437	StackElem stack[1000];
	2442	while (sp > 0) {
non-returning function call therefore sp < 1000	2444	if (sp >= 1000) panic ("stack overflow in qSort3");
	2446	{ sp--; /*...*/ d = *stack[sp].dd*; };
{ sp < 999 }		
	2483	if (gtHi < ltLo) {
{ sp < 999 } so (a) sp < 1000 is [ok];	2484	{ *stack[sp]*.ll = lo; /*...*/ sp++; };
{ sp < 1000 }	2485	continue; // to while (sp > 0) at 2442
	2486	} // end if gtHi < ltLo
{ sp < 999 } so (a) sp < 1000 is [ok];	2494	{ *stack[sp]*.ll = lo; /*...*/ sp++; };
{ sp < 1000 }	2495	{ stack[sp].ll = n+1; /*...*/ sp++; };
{ sp < 1001 } !		
	2496	{ stack[sp].ll = m; /* ...*/ sp++; };
	2497	}
	2498	} // end qSort3

(continued from page 8)

`code`. This means that the call to `hbAssignCodes()` on line 1868 is safe and no additional runtime guarantees are required. This is an ideal outcome because no additional code needs to be introduced that would introduce additional runtime overhead. A cut-down version of a function (`qSort3()`) that cannot be guaranteed to be safe is shown in Table 5.

This function shows a number of subscripting operations on lines 2446 through 2496 (after macro expansion). The control flow of the function permits a compile-time analysis of the min-max range of the subscript `sp`. This analysis shows that the subscripting is valid at lines 2446 through 2495, but a potential buffer overflow exists at line 2496.

As a result, it is necessary to modify the code so that a check is inserted prior to line 2496 to ensure that `sp` is a valid subscript for `stack`. Alternatively, the bound for `stack` can be increased to 1001 at line 2437.

Programmers may sometimes dismiss concerns about buffer overflows in "corner cases that wouldn't happen in real situations." However, software security requires that developers anticipate the actions of malicious users who will search for corner cases like these that can be successfully exploited.

Summary

Source-code audits have been used successfully to identify and remove software flaws from C and C++ programs that otherwise may have resulted in exploitable software vulnerabilities. However, these audits are often imperfect, unstructured, and dependent on the tenacity and knowledge of the auditor.

Safe-Secure C/C++ (SSCC) is a set of methods to eliminate vulnerabilities resulting from buffer overflows and other programming errors

A formal, structured approach such as the one described in this article can be used to prove the safety of analyzed code. Of course, this manual method is both labor-intensive and prone to human error and could be greatly supplemented by the use of automated tools.

References

[1] Seacord, Robert C. *Secure Programming in C and C++*. Addison-Wesley, 2005, ISBN 0321335724.

[2] Fagan, M.E. "Design and Code Inspections to Reduce Errors in Program Development." IBM System Journal, v. 15 n. 3, 1976, pp. 182–211.

[3] Plum, Thomas and David M. Keaton. "Eliminating Buffer Overflows, Using the Compiler or a Standalone Tool." Published in proceedings of the Workshop on Software Security Assurance Tools, Techniques, and Metrics, Long Beach, California, November 7–8, 2005; https://samate.nist.gov/index.php/Past_Workshops. ❑

PC-lint 8.0 for C/C++

The longest continuously advertised software tool in the history of... mankind.

Presents
Bug of the Month
#1511

```
#include <stdio.h>

typedef const char *cSTR;

struct president
    { cSTR fool() const { return "none"; } };
struct OH : public president
    { cSTR fool() const { return "all"; } };

cSTR assume_office( const president * p )
    { return p->fool(); }

int main()
    {
    OH *p = new OH;
    printf( "we will fool %s of the people.\n",
            assume_office( p ) );
    return 0;
    }
```

A new Office Holder OH assumes power hoping to fool all of the people but instead fools no one. What is his mistake? Visit our web site at **www.gimpel.com**

PC-lint for C/C++ will catch this and many other bugs. It will analyze a mixed suite of C and C++ modules to uncover bugs, glitches, quirks and inconsistencies.

Not your Grandpa's lint: PC-lint has introduced several spectacular and revolutionary innovations in the art of static program analysis. Taking clues from initializers, assignments, and conditionals, variable and member values are tracked, enabling reports on potential uses of null pointers and out-of-bounds subscripts.

New with Version 8: Interfunction value tracking – Actual argument values are used to initialize parameters; return values are computed; a multi-pass operation (you control the number of passes) allows you to plumb the depths of function behavior to arbitrary levels.

Plus Our Traditional C/C++ Warnings: Uninitialized variables, inherited non-virtual destructors, strong type mismatches,ill-formed macros, inadvertent name-hiding, suspicious expressions, etc., etc.

Full Language Support for ANSI/ISO C and C++.

PC-lint for C/C++ *$239*
Numerous compilers/ libraries supported. Runs on Windows, MS-DOS, and OS/2.

FlexeLint for C/C++
The same great product for other operating systems. Runs on all UNIX systems, VMS, mainframes, etc. Distributed in shrouded C source form. Call for pricing.

30 Day Money Back Guarantee

Gimpel Software

Serving the C/C++ Community for 20 Years.

CALL TODAY (610) 584-4261 Or FAX (610) 584-4266

www.gimpel.com

PC-lint and FlexeLint are trademarks of Gimpel Software