

Secure Coding in C and C++

An interview with Robert Seacord, senior vulnerability analyst at CERT



INTERVIEW BY IBRAHIM HADDAD

■ Robert C. Seacord, a senior vulnerability analyst at the CERT/Coordination Center at Carnegie Mellon University, has just published the book *Secure Coding in C and C++* (Addison-Wesley, 2005). I sat down with him to discuss software security in the Linux environment and elsewhere.

LWM: *There's an ongoing debate over whether Linux is more secure than Windows. Some people argue that since Linux's source code is freely available, it makes it easy for hackers to implement hacks and break into Linux systems, whereas this becomes more difficult with proprietary operating systems. What's your take on this topic?*

RCS: I agree there's no real security through obscurity, but obscurity does provide an impediment. Attackers can "reuse" sections of code to develop exploits that have the same properties as the code they're attacking — an unintended but unfortunate consequence of reuse.

A number of security experts have argued that Open Source software is more secure because it's open to review by a broad range of individuals. However, Open Source software provides no guarantee that the software is reviewed or that those reviewing it understand the program's context in a larger system or the program's external



interactions. In other words, just being able to read the code doesn't mean you have the wherewithal to do anything with it or about it.

On the other hand, attackers have also become adept at reverse-engineering executables so not releasing the source only slows attackers down. An analysis of CERT/CC vulnerability reports conducted by Omar Alhazmi at Colorado State University shows vulnerabilities accruing in both Windows and Linux operating systems at similar rates.

LWM: *In your experience, which programming languages (e.g., C, C++, Java) provide the most secure programming safeguards and the most tools to ensure the code doesn't contain vulnerabilities?*

RCS: While languages like Java that have automatic garbage collection, lack pointers, and have strict type checking can limit or prevent vulnerabilities resulting from buffer overflows and common dynamic memory management errors, programming errors that result in security vulnerabilities can happen in any language. There are tradeoffs that have to be considered in language and application development platforms that can't always be objectively reduced to "Which is more secure?"

Java programs can be vulnerable to SQL injection and cross-site scripting (XSS) vulnerabilities. Integer errors can also occur undetected, although the consequence of these errors is typically not as severe as they may be in C and C++ language programs.

Interestingly, Java language security may be due more to the reasons given above and less to Java bytecodes than typically understood. C++ programs compiled to MSIL in the Microsoft .NET environment are as susceptible to buffer overflows and other common vulnerabilities as their compiled counterparts, although additional security can be gained by using the new system-level data structures for arrays, strings, and other data types.

Because of the history of security vulnerabilities in C and C++, a number of tools and products have been developed to make these languages more secure. So, ironically, these languages generally have the best tool support.

LWM: *What's the most important coding practice that we can implement to develop more secure software? Can you also provide us your Top 5 best practices for writing secure C/C++ code?*

RCS: The most important coding practice is to be extremely paranoid in handling any data that originated with the user, either directly or indirectly.

As for the remaining four:

1. Many vulnerabilities in C/C++ result from the incautious manipulation of strings. Your entire development team should select a clear and unambiguous

ABOUT THE INTERVIEWER

Ibrahim Haddad is a Strategic Program Manager at the Open Source Development Labs (OSDL), leading the Carrier Grade Linux Initiative and the Mobile Linux Initiative, promoting the development and adoption of Linux in the communication industry. Prior to joining OSDL, Ibrahim was a senior researcher in the "Research and Innovation" Department of Ericsson Corporate Unit of Research, where he was involved with the server system architecture for 3G wireless IP networks. He is currently a Doctoral of Science Candidate at Concordia University in Montreal, Canada, researching "Scalable Architectures for High-Availability Web Server Clusters."

ibrahim@osdl.org

approach to dealing with strings in your application and apply it consistently. C++ programmers have the option of using the standard `std::string` class, which is generally less error prone than standard C-style strings. C language programmers may want to consider using or building a string library so that all string operations (and potential vulnerabilities) can be isolated to a single module. CERT has begun developing a managed string library that could be used as the basis of such an effort and submitted it as a paper to the J11 WG14 C standard working group. Eventually we hope to release the source code for this library on the Secure Coding web site at <http://www.cert.org/>.

2. Your entire development team should select a clear and unambiguous approach to dealing with integers and apply it consistently. Operations on integers can result in overflow, truncation, sign errors, and other related issues that can lead to exploitable vulnerabilities. Having a plan to deal with integers up-front is critical because almost every integer operation can result in an exceptional condition, particularly when the inputs can be controlled by an attacker. Generally speaking, you should limit all integer inputs to acceptable values and use safe integer operations that detect and report error conditions when dealing with “tainted” inputs.
3. Sit down together as a development team and do code inspections. Maintain a list of common programming defects you find and make sure you check for these until they cease to be found. The inspections serve several purposes. First, they can be effective in identifying and removing defects in the code that can lead to vulnerabilities. Second, and potentially more importantly, it lets more experienced coders mentor less experienced team members about what to look for and how to correct problems. Third, inspections provide a mechanism for fostering a consistent approach to applying security-coding practices throughout the project.
4. Use the defense-in-depth approach of applying multiple strategies so that a single error isn't necessarily fatal. Start with secure coding practices and then evaluate your code using a variety of

manual processes and automated tools. Use a secure runtime environment as a final line of defense.

LWM: What are the three most dangerous C lib functions to use and why?

RCS: I would have to say `gets()`, `strcpy()`, and `sprintf()`.

The `gets()` function is on the list because it can't be used safely. The function reads a line from standard input into the buffer until a terminating new line or EOF is found. In fact, the Linux man page for this function contains the following advice:

Because it is impossible to tell without knowing the data in advance how many characters `gets()` will read, and because `gets()` will continue to store characters past the end of the buffer, it is extremely dangerous to use.

There are two alternative functions that can be used: `fgets()` and `gets_s()`. The following example shows how these calls are used:

```

1. #define BUFFSIZE 8
2. int main(int argc, char *argv[]){
3. char buff[BUFFSIZE];
   // insecure use of gets()
4. gets(buff);
5. printf("gets: %s.\n", buff);
   // more secure use of fgets()
6. if (fgets(buff, BUFFSIZE, stdin) == NULL) {
7. printf("read error.\n");
8. abort();
9. }
10. printf("fgets: %s.\n", buff);
   // more secure use of gets_s()
11. if (gets_s(buff, BUFFSIZE) == NULL) {
12. printf("invalid input.\n");
13. abort();
14. }
15. printf("gets_s: %s.\n", buff);
16. return 0;
17. }

```

The `fgets()` function is defined in C99 and has similar behavior to `gets()`. The `fgets()` function accepts two additional arguments: the number of characters to read and an input stream. The `gets_s()` function is defined in ISO/IEC TR 24731 to provide a compatible version of `gets()` that was less prone to buffer overflow.

The `strcpy()` function is on the list because, even though it can generally be used safely, it's often used in an insecure fashion, for example by dynamically allocating the required storage as shown below:

```

1 int main(int argc, char *argv[]) {
2 char *buff = (char *)malloc(strlen(argv[1])+1);
3 if (buff != NULL) {
4 strcpy(buff, argv[1]);
5 printf("argv[1] = %s.\n", buff);
6 }
7 else {
   /* Couldn't get the memory - recover */
8 }
9 return 0;
10 }

```

There are also many, many alternatives to using `strcpy()` that are generally less error prone, including `strcpy_s()`, `strncpy()` (available for many flavors of Unix but not GCC/Linux), `strdup()`, and others.

Finally `sprintf()` is a triple threat. Incautious use of this function can result in a buffer overflow vulnerability if, for example, an attacker provides a string argument for the user variable below that exceeds 495 bytes (512 bytes – 16 character bytes – 1 null byte):

```

1. char buffer[512];
2. sprintf(buffer, "Wrong command: %s\n", user);

```

Secondly, because the `sprintf()` function accepts a formatted output function that accepts a format string and variable number of arguments, it's subject to format string exploits.

Third, if you Google for `sprintf()` on the Internet you can usually find some code that looks like this code (that I found in the first link I selected) from the Linux kernel mailing list at <http://lkml.org/>:

```

int i;
ssize_t count = 0;

for (i = 0; i < 9; ++i)
    count += sprintf(buf + count, "%02x ", ((u8 *)&sl-
reg_num)[i]);

count += sprintf(buf + count, "\n");

```

So what's wrong with this code? Well, `sprintf()` can (and will) return -1 on error conditions such as an encoding error. In this case, the `count` variable, already at zero, can be decremented further — almost always with unexpected results. While this particular error isn't commonly associated with software vulnerabilities, it can easily lead to abnormal program termination.

LWM: Does gcc or Visual Studio produce more secure executables? How do you assess this?

RCS: In general both compilers are constrained by conformance to C language standards such as ISO/IEC 9899. In some places, Microsoft intentionally disregards strict conformance to improve security, for example, by disallowing the %n conversion specifiers for formatted input/output functions in the 2005 version of Visual C++.

I think the most interesting area for differentiation from a security perspective is in each implementation's handling of integers. In a perfect world, C and C++ compilers would identify the potential for exceptional conditions to occur at runtime and provide a mechanism (such as an exception, trap, or signal handler) for applications to handle these events. Unfortunately, the world we live in is far from perfect.

The Visual C++ .NET 2003 compiler generates a compiler warning (C4244) when an integer value is assigned to a smaller integer type. At warning level 1, a warning is issued if a value of type `__int64` is assigned to a variable of type `unsigned int`. At warning level 3 and 4, a "possible loss of data" warning is issued if an integer type is converted to a smaller integer type. For example, the assignment in the following example is flagged at warning level 4:

```
// C4244.cpp
// compile with: /W4
int main() {
    int b = 0, c = 0;
    short a = b + c; // C4244
}
```

Visual C++ .NET 2003 also provides runtime error checks that are enabled by the `/RTC` flag. The `/RTCc` compiler flag, in particular, provides a similar function to compiler warning C4244 by reporting when a value assigned to a smaller data type results in a loss of data. Visual C++ also includes a `runtime_checks` pragma that disables or restores the `/RTC` settings, but it doesn't include flags for catching other runtime errors such as overflows. Visual C++ 2005 adds the ability to catch overflows in `operator::new` (and is on by default).

Runtime error checks aren't valid in a release (optimized) build for performance reasons.

The gcc and g++ compilers include an `-ftrapv` compiler option that provides limited support for detecting signed integer exceptions at runtime. According to the gcc man page, this option "generates traps for signed overflow on addition, subtraction, and multiplication operations." In practice, this means that the gcc compiler generates calls to existing library functions rather than generating assembler instructions to perform these arithmetic operations on signed integers. These are enforced at runtime even when optimization is enabled.

If you use this feature, make sure you use gcc version 3.4 or later because the checks implemented by the runtime system before this version don't adequately detect all overflows and shouldn't be relied on to do so.

Neither compiler passes an argument or byte count on calls to variadic functions implemented using the ANSI stdargs, although it's permitted by the C99 specification and would make variadic functions such as the formatted input/output functions more secure.

LWM: Are there any security issues that are unique to Linux/gcc? How can they be overcome? Are there any solutions in sight?

RCS: Data pointers are used in C and C++ to refer to dynamically allocated structures, call-by-reference function arguments, arrays, and other data structures. An attacker can modify these data pointers (when exploiting a buffer overflow vulnerability, for example). If a pointer is subsequently used as a target for an assignment, an attacker can control the address to modify other memory locations with a technique known as an arbitrary memory write.

Most Linux implementations contain a large number of suitable targets for arbitrary memory writes — addresses that can be overwritten and then used to transfer control to attacker-injected code (or existing code selected by the attacker). Linux uses the executable and linking format (ELF) that uses a global offset table (GOT). The GOT contains the absolute addresses of functions in the executable. An attacker can overwrite a GOT entry for a function with the shellcode address using an arbitrary memory write.

A similar problem exists when an attacker overwrites function pointers directly. The

GCC compiler generates a `.dtors` section in an easily identifiable location that contains destructor functions that are invoked following execution of the main C program. These functions can be overwritten and used to transfer control to arbitrary code even when the destructor functions aren't used in the program!

Arbitrary memory writes can easily defeat canary-based protection schemes. Write-protecting targets is difficult because of the number of targets and because there's a requirement to modify many of these targets (for example, function pointers) at runtime. Buffer overflows occurring in any memory segment can be exploited to execute arbitrary code, so moving variables from the stack to the data segment or heap isn't a solution. The best approach to preventing pointer subterfuge resulting from buffer overflows is to eliminate possible buffer overflow conditions.

One way to limit the exposure from some of these targets is to reduce the privileges of potentially vulnerable processes. OpenBSD, for example, enforces a policy called "W xor X" or "W^X" that requires that no part of the process memory address space is both writable and executable. If implemented on Linux systems, this policy could eliminate some (but not all) targets of arbitrary memory write.

LWM: Are there any other sources of information on secure coding in C/C++ on Linux?

RCS: In addition to my book *Secure Coding in C and C++*, you should also check out *Secure Programming for Linux and Unix HOWTO—Creating Secure Software* from David Wheeler online at <http://www.dwheeler.com/secure-programs>. 📌

LINUXWORLD MAGAZINE WWW.LINUXWORLD.COM

About the Book

Book: *Secure Coding in C and C++*

Author: Robert C. Seacord

Publisher: Addison Wesley Professional

List Price: \$39.99

ISBN: 0321335724

Published: Sep 9, 2005

Pages: 368

Web Site: <http://www.awprofessional.com/title/0321335724#>