

Overflows, underflows, and other security flaws

SECURE PROGRAMMING

Our security guy looks at software tools that you can use to audit and secure your software. **BY KURT SEIFRIED**

One of the consistent themes in Peter Seibel's book *Coders at Work* [1] is: How do you get to be a better programmer? This idea interests me greatly because, in the security world, I see people consistently making the same types of mistakes over and over again.

After a while, you would think that people would learn how to create a temporary file securely. Or perhaps they could at least learn to create, access, and delete buffers in memory safely; however, this is obviously not the case for many programmers (including some very good ones). So, the obvious answer is to look to the people who are writing secure code and emulate and learn from them. But how do you do this, espe-

cially if you have very little time and virtually no budget?

Many programmers like to share, and the beauty of knowledge and information is that I can teach you what I know and I get to keep it as well. Luckily, some programmers have taken the time to package their knowledge and wisdom into software packages; indeed, the number of libraries available to programmers now is staggering (most of us will never have to write an HTTP client or HTML parser thanks to these people). So, what software can you use to write better and more secure software?

Valgrind

Valgrind [2] is probably the most mature open source (GPL-licensed) set of tools for source code and binary

analysis, not just for security issues but for performance issues (the two often go hand in hand). Probably the component that has the most effect on finding and addressing security issues is memcheck.

The memcheck module will find memory that is accessed when it should not be, uninitialized values that are used in dangerous ways, memory leaks (always a potential denial-of-service problem), and bad freeing of heap blocks (double frees, mismatched frees), and it can detect when your program passes overlapping source and destination memory blocks.

Installing and Using Valgrind

Installing Valgrind is trivial on RPM- and dpkg-based systems:

```
yum install valgrind
```

or

```
apt-get install valgrind
```

Compiling Valgrind from source code is also easy:

```
cd valgrind
./autogen.sh
./configure --prefix=/path
make
make install
```

Using Valgrind

Here is where things get a little tricky. Valgrind creates a lot of output. Some of it is false positives (or stuff that isn't too dangerous), and separating the chaff from the wheat can take some work. This leads directly to Debian Bug Report Log 363516 [3] and my article on the security flaw that was created by this source code change [4]. In a nutshell, Valgrind was warning about the use of uninitialized memory, and instead of using a suppression file (to make Valgrind stop complaining) or simply sitting down and getting a really good understanding of the code, the developer chose to simply comment out the offending lines. Unfortunately, this resulted in a predict-

able pool of entropy that ultimately resulted in keys generated by OpenSSL on Debian that were entirely predictable and easy to guess because so few possibilities existed. As is the case with many powerful tools, if they are used without full understanding, the consequences can be severe. An excellent pair of videos from SecurityTube [5] covers some of the basics of using Valgrind on Linux.

If you're like me and you have to program, but don't really trust yourself to program safely, or you don't have the expertise to use programs like Valgrind to truly ensure that your programs are safe, what can you do?

Use a Safer Language

A relatively simple solution addresses most of the issues around memory management, such as allocating memory properly (using uninitialized memory, etc.), using memory safely (buffer overflows, underflows, etc.), and ensuring that memory is destroyed correctly (double frees, memory leaks, etc.): Use a programming language that has built-in memory management (e.g., Python, Java, Perl, etc.). By doing so, you largely avoid problems like having to care or know about the length of a string or an array when you create it. Simply create the string or the array and shove data into it. The buffer is expanded as needed, and when you read from it, you can't go beyond the end of the string because the program interpreter will simply return an error saying there is no more data or items for you to read (as opposed to cheerfully reading from random areas of memory). The downside of course is that some problems and programs don't lend themselves well to these languages (almost all implementations of these languages are in C, which is a memory management nightmare).

Source Code Auditing Tools: PyChecker

Additionally, being a belt and suspenders Unix kind of guy, I use a simple but effective Python source code checker called PyChecker.

PyChecker [6] is a Python source code auditing tool. Although the Python interpreter will catch many programming errors (and raise an exception, print out an error, then exit), it won't catch everything. The use of a global variable

within a class, as opposed to a self variable (which exists only within that instance of the class and is thus much safer in a threaded environment), can raise all kinds of problems and race conditions that are not much fun to track down. However, PyChecker will catch these and let you know about them immediately. Installing PyChecker is easy:

```
easy_install PyChecker
```

If this doesn't work (I got an error), you can manually install it by downloading the PyChecker tarball, unpacking it, and running the install manually:

```
wget http://download.site/2
pychecker-0.8.18.tar.gz
tar -xf pychecker-0.8.18.tar.gz
cd pychecker-0.8.18
python setup.py install
```

Learning to Program Securely

Even with all this, I haven't yet addressed the education angle. If you want to program securely, you'll need to understand, or at least know about, the various types of flaws that make software insecure. These range from the simple, such as the various kinds of buffer overflows, to the esoteric "Use of a Non-reentrant Function in an Unsynchronized Context." The Common Weakness Enumeration (CWE) [7] project is designed to do exactly this; that is, come up with a complete taxonomy of software security flaws with descriptions, information about solutions, and examples of the flaw. Some of the CWE entries have code examples, and unfortunately, most entries lack solid information on how to actually address the problem or avoid it. But as G.I. Joe says, "Now you know, and knowing is half the battle."

The next step of course is learning the mechanics and mindset of secure programming. Although literally dozens of books are available now, some of which are actually quite good, I still prefer the online resources. One of the best and most comprehensive documents is David A. Wheeler's *Secure Programming for Linux and Unix HOWTO* [8].

Another project that is attempting to improve software security is the Open

Web Application Security Project (OWASP) [9]. Although aimed at web applications, most of the content in the Development Guide, the Code Review Guide, and the Testing Guide apply to software that is not web based. Most importantly, they include actual tools and specific documentation on how to program securely, in addition to "Web-Goat," a deliberately insecure application that allows you to learn from the mistakes of others.

Conclusion

Secure programming isn't very hard. Mostly, it requires discipline. This means no cutting corners, taking the time to understand what your code changes will do (especially if you didn't write the code originally), and most importantly, knowing how your code interacts with other code and systems. ■

INFO

- [1] Seibel, Peter. *Coders at Work*. Apress, 2009, <http://www.codersatwork.com/>
- [2] Valgrind: <http://valgrind.org/>
- [3] Debian Bug report logs – #363516: <http://bugs.debian.org/cgi-bin/bugreport.cgi?bug=363516>
- [4] "Crash Investigation" by Kurt Seifried, *Linux Magazine*, August 2008, pg. 70
- [5] SecurityTube: <http://www.securitytube.net/Profiling-Programs-for-Security-Holes-with-Valgrind-video.aspx>
- [6] PyChecker: <http://pychecker.sourceforge.net/>
- [7] Common Weakness Enumeration: <http://cwe.mitre.org/data/slices/2000.html>
- [8] *Secure Programming for Linux and Unix HOWTO*: <http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO.html>
- [9] OWASP: <http://www.owasp.org/>

THE AUTHOR

Kurt Seifried is an Information Security Consultant specializing in Linux and networks since 1996. He often wonders how it is that technology works on a large scale but often fails on a small scale.

