Make your software behave: Learning the basics of buffer overflows
Get reacquainted with the single biggest threat to software security

Gary McGraw and John Viega
Reliable Software Technologies
March 1, 2000

In our previous column, we discussed software security analysis at a high level, introduced a methodology for assessing software security, and explained the key characteristics of a risk analysis. One essential aspect of any risk analysis is knowing the most common risks and how they are likely to be introduced. Part of this knowledge includes familiarity with the things that coders have a fair chance of doing wrong and that almost always lead to security problems. As a consequence, in this column, we'll introduce the single biggest software security threat: the dreaded buffer overflow.

This week's column includes material originally written by Tom O'Connor, Software Engineer at Surety.Com.

Buffer overflows have been causing serious security problems for decades. In the most famous example, the Internet worm of 1988 used a buffer overflow in fingerd to exploit tens of thousands of machines on the Internet and cause massive headaches for server administrators around the country; see Resources later in this column. But the buffer overflow problem is far from ancient history. Buffer overflows accounted for over 50 percent of all major security bugs leading to CERT/CC advisories last year. (The CERT/Coordination Center is part of the Software Engineering Institute in Pittsburgh; see Resources.) And the data show that the problem is growing instead of shrinking; see "Buffer overflow: Déjà vu all over again".

Clearly, you would think by now that buffer overflow errors would be obsolete. So why are buffer overflow vulnerabilities still being produced? Because the recipe for disaster is surprisingly simple. Take one part bad language design (usually in C and C++), mix in two parts poor programmer practice, and you have a recipe for big problems. Buffer overflows can happen in languages other than C and C++, though without some incredibly unusual programming, modern "safe" languages like Java are immune to the problem. In any case, legitimate reasons often justify the use of languages like C and C++, and so learning their pitfalls is important.

The root cause of buffer overflow problems is that C (and its red-headed stepchild, C++) is inherently unsafe. There are no bounds checks on array and pointer references, meaning a developer has to check the bounds (an activity that is often ignored) or risk encountering problems. A number of unsafe string operations also exist in the standard C library, including:

- strcpy()
- strcat()
- sprintf()
- gets()

For these reasons, it is imperative that C and C++ programmers who are writing security-critical code educate themselves about the buffer overflow problem. The best defense is a good education on the issues. This is why our next four columns will deal with buffer overflow. This column gives an overview of the buffer overflow problem. The next column covers defensive programming techniques (in C), and explains why certain system calls are problematic and what to do instead. In the final two columns in this series, we'll examine the engine's workings and explain how a buffer overflow attack does its dirty work on particular architectures.

What is a buffer overflow?

Buffer overflows begin with something every program needs: a place to put bits. Most computer programs create sections in memory for information storage. The C programming language allows programmers to create storage at run-time in two different sections of memory, the stack and the heap. Generally, heap-allocated data are the kind you get when you malloc() or new something. Stack-allocated data usually include non-static local variables and any parameters passed by value. Most other things are stored in global static storage. (We'll cover these nitty-gritty details two columns from now.) When contiguous chunks of the same data types are allocated, the memory region is known as a buffer.

When writing to buffers, C programmers must take care not to store more data in the buffer than it can hold. Just as a glass can only hold so much water, a buffer can only hold so many bits. If you put too much water in a glass, the extra water has to go somewhere. Similarly, if you try to put more data in a buffer than fits, the extra data have to go somewhere, and you might not always like where it goes!

When a program writes past the bounds of a buffer, this is called a buffer overflow. When this happens, the next contiguous chunk of memory is overwritten. Since the C language is inherently unsafe, it allows programs to overflow buffers at will (or, more accurately, completely by accident). There are no run-time checks that prevent writing past the end of a buffer, so a programmer has to perform the check in his or her own code, or run into problems down the road.

Reading or writing past the end of a buffer can cause a number of diverse (and often unanticipated) behaviors: 1) programs can act in strange ways, 2) programs can fail completely, or 3) programs can proceed without any noticeable difference in execution. The side effects of overrunning a buffer depend on:

- How much data are written past the buffer bounds
- What data (if any) are overwritten when the buffer gets full and spills over
- Whether the program attempts to read data that are overwritten during the overflow
- What data end up replacing the memory that gets overwritten

The indeterminate behavior of programs that have overrun a buffer makes them particularly tricky to debug. In the worst cases, a program may be overflowing a buffer and not showing any adverse side effects at all. As a result, buffer overflow problems are often invisible during standard testing. The important thing to realize about buffer overflows is that any data that happen to be allocated near the buffer can potentially be modified when the overflow occurs.

Why are buffer overflows a security problem?
You may be thinking, "Big deal, a little spilled water never hurt anybody." To stretch our analogy a bit, imagine that the water is spilling on a worktable with lots of exposed electrical wiring. Depending on where the water lands, sparks could fly. Likewise, when a buffer overflows, the excess data may trample on other meaningful data that the program might wish to access in the future. Sometimes, changing these other data can lead to a security problem.

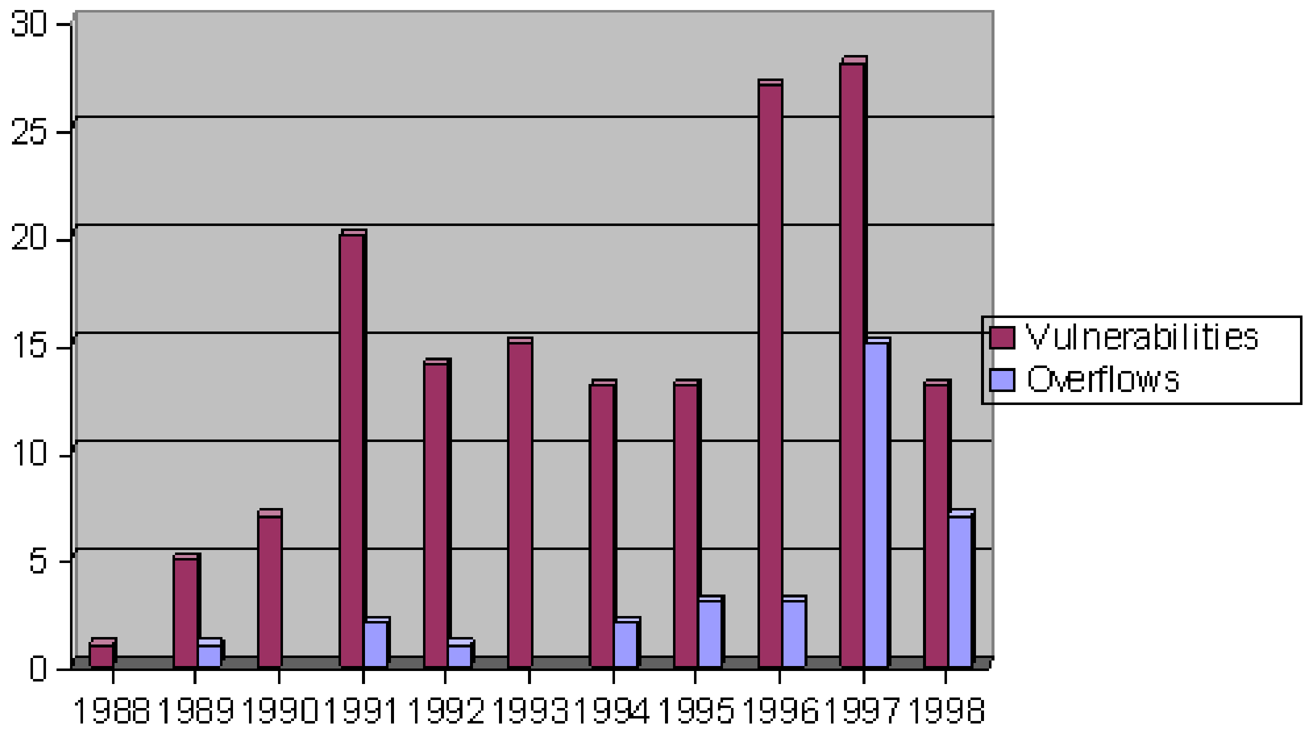
In the simplest case, consider a Boolean flag allocated in memory directly after a buffer. Say that the flag determines whether or not the user running the program can access private files. If a malicious user can overwrite the buffer, then the value of the flag can be changed, thus providing the attacker with illegal access to private files.

Another way in which buffer overflows cause security problems is through stack-smashing attacks. Stack-smashing attacks target a specific programming fault: careless use of data buffers allocated on the program's run-time stack, namely local variables and function arguments. The results of a successful stack-smashing attack can be far more serious than just flipping a Boolean access control flag as in the previous example. A creative attacker can take advantage of a buffer overflow vulnerability through stack-smashing and then run arbitrary code (anything at all). The idea is pretty straightforward: Insert some attack code (for example, code that invokes a shell) somewhere and overwrite the stack in such a way that control gets passed to the attack code. (We'll go into the details of stack smashing in our third and fourth columns on buffer overflows.)

Commonly, attackers exploit buffer overflows to get an interactive session (shell) on the machine. If the program being exploited runs with a high privilege level (such as root or administrator), then the attacker gets that privilege in the interactive session. The most spectacular buffer overflows are stack smashes that result in a superuser, or root, shell. Many exploit scripts that can be found on the Net (see Resources) carry out stack-smashing attacks on particular architectures.

Buffer overflow: Déjà vu all over again
Up to 50 percent of today's widely exploited vulnerabilities are buffer overflows, according to an analysis by David Wagner, Jeffrey Foster, Eric Brewer, and Alexander Aiken in a paper presented at this year's Network and Distributed Systems Security conference (NDSS2000). Furthermore, the analysis suggests that the ratio is increasing over time. The data are extremely discouraging since the buffer overflow problem has been widely known in security circles for years. For some reason, developers have not readily moved to eliminate buffer overflows as the leading pitfall in software security.

**Number of vulnerabilities resulting in CERT/CC advisories
for the last eleven years**



In chart above, the number of vulnerabilities that can be directly attributed to buffer overflows is displayed. As the data show, the problem is not getting any better. In fact, buffer overflows are becoming more common.

Heap overflows versus stack overflows
Heap overflows are generally much harder to exploit than stack overflows (although successful heap overflow attacks do exist). For this reason, some programmers never statically allocate buffers. Instead, they malloc() or new everything, and believe this will protect them from overflow problems. Often they are right, because there aren't many people who have the expertise required to exploit heap overflows. But dynamic buffer allocation is not intrinsically less dangerous than other approaches. Don't rely on dynamic allocation for everything and forget about the buffer overflow problem. Dynamic allocation is not a cure-all.

Let's dig deeper into why some kinds of buffer overflows have big security implications. A number of interesting UNIX applications need special privileges to accomplish their jobs. They may need to write to a privileged location like a mail queue directory, or open a privileged network socket. Such programs are generally suid (set uid) root, meaning that the system extends special privileges to the application upon request, even if a regular old user runs the program. In security, anytime privilege is being granted (even temporarily) there is potential for privilege escalation to occur. Successful buffer overflow attacks can thus be said to be carrying out the ultimate in privilege escalation. Many well used UNIX applications, including lpr, xterm and eject, have been abused into giving up root through exploit of buffer overflow in suid regions of the code.

A common cracking technique is to find a buffer overflow in an suid root program, and then exploit the buffer overflow to snag an interactive shell. If the exploit is run while the program is running as root, then the attacker will get a root shell. With a root shell, the attacker could do pretty much anything, including viewing private data, deleting files, setting up a monitoring station, installing back doors (with a root kit), editing logs to hide tracks, masquerading as someone else, breaking stuff accidentally, and so on. Very bad.

Meanwhile, many people believe that if their program is not running suid root, they don't have to worry about security problems in their code, since the program can't be leveraged to achieve greater access levels. That idea has some merit, but is still a risky proposition. For one thing, you never know who is going to take your program and set the suid bit on the binary. When people can't get something to work properly, they get desperate. We've seen this sort of situation lead to entire directories of programs needlessly set setuid root. Once again, very bad.

There can also be users of your software with no privileges at all. That means any successful buffer overflow attack will give them more privileges than they previously had. Usually, such attacks involve the network. For example, a buffer overflow in a network server program that can be tickled by outside users may provide an attacker with a login on the machine. The resulting session has the privileges of the process running the compromised network service. This type of attack happens all the time. Often, such services run as root (and generally for no good reason other than to make use of a privileged low port). Even when such services don't run as root, as soon as a cracker gets an interactive shell on a machine, it is usually only a matter of time before the machine is "owned" -- that is, the attacker gains complete control over the machine, such as root access on a UNIX box or administrator access on a Windows NT box. Such control is typically garnered by running a different exploit through the interactive shell to escalate privileges.

The extent of the real-world problem
Buffer overflows are, sadly, all too common in C programs. As stated above, even well used and well scrutinized programs are susceptible to them.

For example, Sendmail, an email server and one of the most widely used programs on the Net, is notorious for having security problems. Many people use this software, and many have scrutinized it carefully, but serious security problems (including buffer overflows) continue to be unearthed. In September of 1996, after several new exploitable buffer overflows were found and fixed in Sendmail, an extensive manual security audit was performed. But not four months later, yet another exploitable buffer overflow was discovered that the manual audit missed.

To fast-forward a bit, a graduate student from the University of California in Berkeley, David Wagner, found a handful of new buffer overflows in Sendmail as recently as this year (see Resources). However, the overflows he found do not seem to be exploitable. In other words, it appears that none of the buffers could be overflowed arbitrarily by carefully-crafted attacker input (something that is generally a requirement). Nonetheless, at least one of the overflows Wagner found is known to have survived the manual audit from 1996.

And just because a bug "doesn't seem to be exploitable" by trained security experts doesn't mean that it isn't exploitable. In 1998, researchers at Reliable Software Technologies (not the authors) found three buffer overflow conditions in the Washington University ftp daemon (wu-ftpd) version 2.4 using a fault injection tool called FIST. (We'll revisit FIST later in the series when we talk about dynamic analysis.) This in itself was interesting, because wu-ftpd had recently been heavily scrutinized after four CERT Advisories on wu-ftpd were issued between 1993 and 1995. Experts at Reliable Software Technologies examined the potentially vulnerable code and were unable to learn of a way to exploit any of the conditions. They declared the likelihood was very low that user input could be successfully manipulated to reach a vulnerable buffer in such a way as to cause a security violation. Then about a year later, a new CERT advisory on wu-ftpd appeared. One of the three overflows they had detected turned out to be a vulnerability after all!

These anecdotes show how difficult manual analysis can be. By its very nature, code tends to be complex. Analyzing large programs like Sendmail (approximately 50,000 lines of code) is no easy task. Problems often slip by unwary developers, but problems slip by the experts, too.

Windows versus UNIX: Is security by obscurity good?
So far, all our examples of buffer overflow exploits have been for UNIX systems. But we wouldn't want Microsoft Windows to feel left out! Numerous buffer overflow opportunities exist on Windows machines too. Recently, a company called eEye found an exploitable buffer overflow in Microsoft's Internet Information Server (IIS). This hole left approximately 90 percent of all Web servers running on Windows NT machines wide open for any script kiddie to "own." (Script kiddies are would-be hackers who know enough to run a pre-coded script against a vulnerable machine, but not enough to create such a script in the first place. See the first column, "Make your software behave" for more on script kiddies.)

Some people believe that it's harder to find buffer overflows in Windows programs than in UNIX programs. There is some truth to this, because Windows programs tend to ship without source code, and UNIX programs tend to ship with source code. It's a lot easier to find potential buffer overflows (or most security flaws, for that matter) when you have the source code for a program. When the source code is available, an attacker can look for all suspect function calls, and then try to determine which ones might end up being vulnerabilities. Looking at the program, it is also easier for an attacker to figure out how to cause a buffer overflow with real inputs. It takes a lot more skill (and probably a measure of luck) to do these things without source code.

The security you get for free when you don't release your source code is commonly known as "security by obscurity." It is a bad idea to rely on this kind of security, however, because skilled attackers are still going to be able to break your code if it wasn't designed properly in the first place. A much better solution is to create something solid and let others see what you've done so they can help you keep it secure.

Many Windows developers rely on the security by obscurity model, whether they know it or not. Often, they don't know it, mainly because they never really give a thought to security when developing their applications. In such a case, they are implicitly relying on a flawed approach.

Security experts generally believe that Windows code has more exploitable buffer overflow problems than UNIX code. This reasoning can be attributed to the "many eyeballs" phenomenon, which holds that open source software tends to be better scrutinized, and therefore is more likely to be secure. Of course, there are caveats. First, having people look at your source code doesn't guarantee that bugs will be found. Second, plenty of open source software exists for Windows, and at least as much closed source software exists for UNIX machines. For these reasons, the phenomenon really isn't architecture dependent.

Conclusion
In this column we've introduced you to buffer overflows, which are probably the worst software security problem of all time. One thing we didn't do is give you any advice on how to avoid these problems in your own programs. We also only hinted at the nitty-gritty details of how stack overflows work. You're in luck, though; we'll address these issues in our next column.

Resources

- M. Eichin and J. Rochlis, "With microscope and tweezers: an analysis of the Internet virus of Nov. 1988," 1989 IEEE Symposium on Security and Privacy, Oakland, CA
- The CERT/Coordination Center, part of the Software Engineering Institute in Pittsburgh
- Some exploit scripts that carry out stack-smashing attacks on particular architectures
- D. Wagner, J. Foster, E. Brewer, and A. Aiken, "A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities," 2000 ISOC Network and Distributed Systems Security Symposium (NDSS2000), San Diego, CA
- The first "Make your software behave" column on *developerWorks*, where Gary and John introduce their philosophy of security, and explain why they're focusing on software security issues facing developers
- "Make your software behave: Assuring your software is secure" on *developerWorks*, where Gary and John lay out their five-step process for designing for security

About the authors

Gary McGraw is the vice president of corporate technology at Reliable Software Technologies in Dulles, VA. Working with Consulting Services and Research, he helps set technology research and development direction. McGraw began his career at Reliable Software Technologies as a Research Scientist, and he continues to pursue research in Software Engineering and computer security. He holds a dual Ph.D. in Cognitive Science and Computer Science from Indiana University and a B.A. in Philosophy from the University of Virginia. He has written more than 40 peer-reviewed articles for technical publications, consults with major e-commerce vendors including Visa and the Federal Reserve, and has served as principal investigator on grants from Air Force Research Labs, DARPA, National Science Foundation, and NIST's Advanced Technology Program.

McGraw is a noted authority on mobile code security and co-authored both "Java Security: Hostile Applets, Holes, & Antidotes" (Wiley, 1996) and "Securing Java: Getting down to business with mobile code" (Wiley, 1999) with Professor Ed Felten of Princeton. Along with RST co-founder and Chief Scientist Dr. Jeffrey Voas, McGraw wrote "Software Fault Injection: Inoculating Programs Against Errors" (Wiley, 1998). McGraw regularly contributes to popular trade publications and is often quoted in national press articles.

John Viega is a Senior Research Associate, Software Security Group co-founder, and Senior Consultant at Reliable Software Technologies. He is the Principal Investigator on a DARPA-sponsored grant for developing security extensions for standard programming languages. John has authored over 30 technical publications in the areas of software security and testing. He is responsible for finding several well-publicized security vulnerabilities in major network and e-commerce products, including a recent break in Netscape's security. He is also a prominent member of the open-source software community, having written Mailman, the GNU Mailing List Manager, and, more recently, ITS4, a tool for finding security vulnerabilities in C and C++ code. Viega holds a M.S. in Computer Science from the University of Virginia.

What do you think of this article?

○ Killer! ○ Good stuff ○ So-so; not bad ○ Needs work ○ Lame!

Comments?

Submit feedback