Global Information Assurance Certification Paper

"Buffer Overflows: Why, How and Prevention"
Nicole LaRock Decker

In November 1988 an incident occurred that caught the attention of the entire networked community - the Internet Worm. The Internet Worm brought down more than 6000 sites by a little-known exploit called the buffer overflow [1]. Pretty powerful, huh? To this date, buffer overflows are still being found in a variety of software in a variety of operating systems. They are a serious threat to any system or network administrator's well-being. It is a problem that has been around for way too long and needs to be seriously addressed. This paper focuses on why buffer overflows are still around today, how they work, and provides a list of some ways to help prevent and protect.

After the Internet Worm, we didn't hear much about buffer overflows again until the mid to late '90s when a series of documents were published and released on the 'Net. They were:

1. "Smashing The Stack For Fun and Profit" written by "Aleph One," published in the Phrack magazine in November of 1996.
2. "How to Write Buffer Overflows" by "Mudge" released in 1997.
3. "Stack Smashing Vulnerabilities in the UNIX Operating System" by Nathan P. Smith. also released in 1997.
4. "The Tao of Windows Buffer Overflows" by "DilDog," April of 1998.

In each of these papers, several items were addressed - what a buffer overflow is, how they work, how to find one, how to create one and each paper even provided sample code for gaining root access to a system once an overflow was found. The first three papers concentrated on buffer overflows in the UNIX operating system. The last paper focused on buffer overflows in the Windows operating environment. So, in a matter of a year and a half, critical information in exploiting holes in 2 major operating systems was released. Today, there are even more articles on the 'Net talking about the ins and outs of buffer overflows, allowing attackers to get even more perspectives on the exploit. However, it was the release of these first few papers that really caught the attention of system and network administrators as well as attackers looking for ways into systems and networks.

The existence of the above documents could help explain why buffer overflows have been so popular within the last few years. To get an idea of the abundance of buffer overflows, let's look at the number of vulnerabilities involving buffer overflows from 1998 to the present, November 2000. The following numbers are derived from research I conducted based on information from the CERT Coordination Center (CERT/CC), Common Vulnerabilities and Exposures (CVE), and Security Focus web sites. Searching for the term "buffer overflow" in the title or description of the CERT/CC Advisories returned the following. Of the 1998 CERT/CC advisories, 7 out of 13 advisories were due to buffer overflows, about 54%. In 1999, there were 9 out of 17, about 53%, and in this year, 2000, there have been 2 of 19 thus far, about 11%. So, in 1998 and 1999 over half the advisories were due to buffer overflows - an astounding amount. The fact that there has been a significant reduction for the year 2000 raises some eyebrows. Even though the year isn't over, it still raises some questions. Are buffer overflows finally going down on the list of top

security holes? Are attackers finding other, possibly easier, ways of gaining control of systems? Are there fewer programs susceptible to buffer overflows? Is software code being written more securely? We'll have to wait for the statistics over the next couple of years to find out for sure.

The CVE database, which was started in September of 1999, contains 1077 entries, as of November 2000. Conducting a search for "buffer overflow" in the descriptions of all 1077 entries returned 312 vulnerabilities, about 29%.

Doing a keyword search of "buffer overflow" in the Security Focus advisories found 93 in 1998, 161 in 1999, and 106 for 2000 thus far. This is consistent with the CERT/CC advisories. There was a growth from 1998 to 1999 and then a decrease from 1999 to 2000. Again the same types of questions can be asked as to the cause of this decline from 1999 to 2000. The interesting point though is that, within the last few years, buffer overflows are still found in high numbers. Even though they have been around since the Internet Worm in 1988, they are still, based on the numbers given above, one of the most common exploits. Since most buffer overflows result in root access to a system, they are consequently one of the most lethal exploits. They have been found in many platforms, especially UNIX and Windows based platforms, and have affected a wealth of applications such as, in the case of the Internet Worm, fingerd, Microsoft NetMeeting, Internet Explorer, Microsoft Internet Information Server, ftp, imap, bind, pine, lpr, sendmail, X Windows, ssh, and the list goes on and on.

To help understand these exploits, let's look at how buffer overflows occur. The method we will be describing below is a type of buffer overflow commonly referred to as "stack smashing." To comprehend how stack smashing occurs, we first need to understand a little bit about buffers and instruction pointers. Consider the gas tank in your car. If you have a 15 gallon tank, you can at most fit 15 gallons of gasoline in it. That's your buffer size. If you try to put 17 gallons in it, you've overflowed the tank and wasted your money. The same concept can be applied to variables within a program. For example, say a program requires input from the user that is programmed to be a maximum of 256 bytes. There is a spot reserved in the computer's memory (a buffer) to store these 256 bytes or less. If the user inputs more than 256 bytes and the program does not check that this amount has been exceeded, he or she has overflowed the buffer. Since the computer needs a place to put these extra bytes, it generally puts them in a neighboring spot, overwriting what was already there.

There is another buffer in the computer's memory that stores the memory address of the next command to execute. This is called the instruction pointer. Continuing our example above, after reading the user's input, the next statement might be to print it to the screen. The instruction pointer would have as it's contents the memory address of this print statement. The computer gets the input from the user, stores it in the buffer, checks the instruction pointer to find what to execute next, finds the memory address of the print statement, retrieves the contents of the input buffer, and finally prints the user's input to the screen.

Now, putting the two together, if an attacker were able to overflow the buffer so that it modified the return address of the instruction pointer in such a way that it pointed to the attacker's code, they could do some clever things. And that is exactly what happens. The attacker overflows the buffer so that the memory address for the instruction pointer is the memory address of, in most cases, the function to execute /bin/sh. When the program finishes the part of the program that overflowed the buffer, it finds the address for /bin/sh in the instruction pointer and executes it.

As long as the program is running with root privilege the attacker now has a root shell and has complete control of your system.

The concept of overflowing a buffer sounds pretty straight-forward. As it turns out, overflowing the buffer to modify the return address of the instruction pointer is not a trivial task. It requires knowledge of the C programming language, the assembly language, hexadecimal arithmetic, and a program debugger such as dbx. This may sound like a lot, which it is, but the papers released in 1996-1998 have given people a head start in comprehending all of these items. One paper even points to some tools you can download to assist you. And the proliferation of information released on the Web makes it even easier. All it takes is one person with the knowledge and technical expertise to create the buffer overflow and publish it on the Web before it's in the hands of many attackers.

Many attribute the existence of buffer overflows to poor programming practices, especially those programs written in the C programming language [1]. When programs ask for a maximum size input or copy one array to another, proper provisions should be made to make sure that the size limits are not exceeded. This is called bounds checking. There are library functions that exist in the C programming language that do no bounds checking. Examples include the strcat() and strcpy() functions. When software is released in the Internet community in full source code, attackers can merely search for these types of functions in the code. If they find them, they are on their way to finding an overflow. Now that the Open Software movement is in full-force, this has become especially true.

So why not just write software that does bounds checking, right? Sounds easy enough. Some programmers just aren't aware that they need to do bounds checking. Some don't want to because it may affect the performance speed of the program. Also, it has only been within the last five to six years that buffer overflows have become well-known. There is plenty of legacy software that was written before programmers knew that bounds checking was a must. Lots of this legacy software has not been rewritten to be more secure. Some software isn't rewritten to be more secure until a vulnerability is found. Once buffer overflows are found, most companies are good at releasing patches to fix them, but this still depends on system administrators actually installing the patch. Some system administrators don't patch their systems in a timely fashion - some just don't do it at all.

This is why awareness of the problem and doing all that you can to help prevent buffer overflows is an absolute must. There are a few things that programmers and system and network administrators can do to help prevent and protect systems from buffer overflow attacks. The following is a compilation of suggestions, guidelines, and utilities that will assist in doing just that:

- Get with the program!! (no pun intended) Write programs that do bounds checking.
- Review legacy software code for security problems.
- Avoid functions in C that do no bounds checking. Instead, use their bounds checking substitutes. The following is a recommendation is from "A Lab engineers checklist for writing secure Unix code."

Instead Of:	Use:
gets()	fgets()
strcpy()	strncpy()
strcat()	strncat()
sprintf()	bcopy()
scanf()	bzero()
sscanf()	memcpy(), memset()

- Be careful when using for and while loops that copy data from one variable to another. Make sure the bounds are checked.
- Be especially careful programming and/or installing setuid root programs and programs that run as root. These are the programs that allow an attacker to acquire a root shell.
- There is a compiler called "StackGuard" which if used to compile code will protect against stack smashes. It's "implemented as a small patch to the gcc code generator" [2]. It "seeks not to prevent stack smashing attacks from occurring at all, but rather to prevent the victim program from executing the attacker's injected code. StackGuard does this by detecting that the return address has been altered *before* the function returns" [3].
- Another compiler, called "StackShield" is a stack smashing protection tool for Linux. It "integrates with GCC and basically adds a little bit of code that checks the return address of a function and makes sure it is within the correct limits, if it isn't you can have the program exit." [4]
- There is an operating system called "Immunix OS". The Web site for it says that "Immunix OS 6.2 is based on Red Hat 6.2, but with all C source-available programs re-compiled with the StackGuard compiler. The result is a system that is fundamentally compatible with Red Hat Linux, but is secured against a majority of all Internet security attacks. "
- There is another version of linux designed to be more secure called "Bastille Linux." According to information on the Bastille Linux web site, "The Bastille Hardening System attempts to 'harden' or 'tighten' the Linux operating system. It currently supports Red Hat and Mandrake systems." It also says that it "draws from every available major reputable source on Linux Security."
- The following are utilities and papers that focus on code review:
    - SLINT: Source code Security Analyzer for UNIX and NT from L0pht is a utility that checks your code for potential security holes.
    - Security Code Review Guidelines by Adam Shostack is a document that provides guidelines for writing secure code.

    - "A Lab engineers check list for writing secure Unix code" from AUSCERT is another document providing insight into writing secure code.

- Subscribe to some good security mailing lists such as those for:
    - CERT Advisories
    - SANS Security Digests
    - BUGTRAQ Advisories
- Install security patches ASAP!!

In summary, "buffer overflows" is a term that has become very common in the realm of system and network security. They are a major security threat and should be prevented as much as possible. Programmers and System and Network Administrators need to be more proactive when it comes to writing software and protecting systems. Education and awareness on what can be done is high priority!

References:

[1] Farrow, Rick. "Blocking Buffer Overflow Attacks." Nov 1999. URL: http://www.networkmagazine.com/magazine/archive/1999/11/9911def.htm (2 May 2000).

[2] unknown. "StackGuard Mechanism: Stack Integrity Checking." URL: http://www.cse.ogi.edu/DISC/projects/immunix/StackGuard/mechanism.html (10 Nov. 2000).

[3] Cowan, Crispin, Beattie, Steve, Day, Ryan Finnin, Pu, Calton, Wagle, Perry, Walthinsen, Erik. "Protecting Systems from Stack Smashing Attacks with StackGuard." May 1999. URL: http://www.cse.ogi.edu/DISC/projects/immunix/lexpo.pdf (10 Nov. 2000).

[4] Seifried, Kurt. "Protecting yourself from your software." 3 Nov 1999. URL: http://securityportal.com/direct.cgi?/closet/closet19991103.html" (9 Nov. 2000).