Global Information Assurance Certification Paper

Buffer Overflows and Application Security

Craig Sheppard
February 3, 2004

Abstract

The purpose of this paper is to present in a less-technical fashion an overview of the definition and history of the buffer overflow, its implications for application security and steps that can be taken to safeguard applications. By being less-technical, this presentation is intended to benefit those not directly involved in programming or application development by allowing them an appreciation for the buffer overflow condition and its effect on security. Although any number of computer architectures, operating systems and programming languages are susceptible to buffer overflow exploits, examples in this paper are drawn mostly from the C/C++ programming language and Unix-based operating systems.

Introduction

Jeremiah Grossman lists thirteen web application attack types in his presentation on web application security. They are: "Hidden Value Manipulation, Parameter Tampering, Poison Cookie, Cross Site Scripting, Cookie Theft, Buffer Overflow, Forceful Browsing, Brute Force, Filter Bypass Manipulation, Direct Access Browsing, Back Doors and Debug Options, Known Vulnerabilities, and Erroneous Error Handling" (Grossman, slide 18). Out of all these methods, the attack type most often used (or at least the one most often uncovered) by more than half is the innocuous buffer overflow. As a nuisance it has been around for decades. As an exploit it came into its own in 1988.

In the course of executing, a software program often needs a place to temporarily store data. At these points in the program, the programmer designates a section of memory for storage. When the same kind of data is assigned to contiguous blocks of memory, that region of memory is called a buffer. This is how it happens for most programming languages. And buffers are good. They are a valuable and much-needed part of the program.

In the C programming language, the programmer can designate one of two sections of memory for this type of storage – the stack or the heap (two terms that we'll cover in more detail later). Typically the type of data stored would be either a variable, a parameter value or a return address pointer. Once you define the buffer in the program, you set its bounds (or size) and write the data to it. If what is written exceeds the bounds of the buffer, the buffer overflows. Where the excess data flows to is what makes this an interesting problem. Whereas most programming languages check the bounds of the buffer before writing to it, the C

programming language is notorious for allowing the program to overwrite existing data in the neighboring blocks of memory with new data. C and C++ are two languages that allow this to happen very easily. What happens next is where the fun begins. It depends on how much data overflowed, what data was overwritten, what data it was overwritten by and whether or not the program ever returns to pull the now "missing" or overwritten data from the buffer.

Historical Overview

Of course, none of this is new to C programmers. In the seventies, when buffer overflows were better known as array bounds exceptions (the word 'array' describing the contiguous blocks of memory), Henry Baker "spent quite a number of unpaid overtime nights debugging "array bounds exceptions" from "core dumps" to avoid the even worse problems which would result from not checking the array bounds." (Baker, p.1). Notice the nuisance factor here. Long before they became a security problem, buffer overflows were dealt with as a function of getting the program to work. Henry's mention of core dumps (a complete listing of everything in memory and/or the cpu registers) illustrates the difficulty of troubleshooting buffer overflows. You had to study what was in memory and then backtrack to find out where it was in the program and what data it was that had gone over its bounds. In those days when processing power was more precious than people power, programmers were discouraged from including buffer checks in their own code for reasons of space and efficiency. They were expected to reasonably predict the data storage requirements for any of the information they wanted to park in the buffer. Parameter values being passed from other sub-routines added some complexity but were still manageable, so the arrays were often left unchecked. Even today, when processing power and disk space are so much more inexpensively abundant, the old programming mindset still exists.

The feeling of safe harbor shared amongst members of the networked community dissipated in 1988 when Robert Morris released a worm onto the net. In this one program the buffer overflow was successfully exploited to a remarkable degree – even to the amazement of its creator. 6000 systems, or one-tenth of the machines on the net then, were affected. It galvanized what were mostly university research teams who united to defeat the worm. Out of the chaos organizations such as the Computer Emergency Response Team (CERT) were created to better prepare the net community for future attacks.

In 1996 the security arena was jolted again when Aleph One published his essay titled "Smashing the Stack for Fun and Profit." In a very matter-of-fact style the article clearly outlines how a program can be compromised via the buffer overflow. The Morris worm showed it could be done. Now the Aleph One article explained how it could be done. The number of buffer overflow exploits have been on the rise ever since.

Implications for Application Security

Now let's examine how a buffer overflow can be used to attack a program. The Morris worm came onto the stage in a big way and is an excellent example. It utilized a buffer overflow exploit in the stack of the finger daemon (or server) program. There are two very similar terms we want to distinguish here. A finger daemon provides information about users. Fingerd provides a network interface to a finger daemon. Someone looking for information about a user, such as a location or phone number, would query a finger server. Let's consider the following four bullet points that Mengzhi Wang presented to her class:

- BSD finger daemon responds to incoming requests by reading data (usually a user id) into a buffer and transmitting finger information back
- The buffer was 512 bytes long, allocated on the stack, and filled using gets:
  ```
  char buffer[512];
  gets(buffer);
  ```
- Malicious request overwrites return address on the stack to point into the buffer, which contains code to exec /bin/sh
- on most systems, fingerd was run as root (Wang, p.1)

The worm program runs fingerd, which listens for a TCP port on a finger daemon. The first bullet shows how a normal finger daemon behaves in response to a request from fingerd. The second bullet details the buffer on the stack of the finger daemon that had been established to handle this query. The finger daemon reads the request from the remote host using the *gets ()* function, which doesn't check the size of the data. The third bullet describes the overflow. In this case the request that the finger daemon receives from fingerd is 536 bytes, exactly 24 bytes larger than the 512 bytes that the finger daemon had allocated. The extra data overwrites the return address pointer in the adjacent memory block. Normally this pointer would return control to the main program when this query was finished. Now there is a new pointer in that location that directs the program back to the buffer. There it finds not the usual "user id" request, but machine code that executes /bin/sh, or a command interpreter. This type of code is commonly referred to as the "egg". The worm has successfully delivered and executed its package and it now has a shell (a user session in Unix, much like a DOS prompt) on the finger server. As bullet four notes, it is probably a root shell. Root is a default user in Unix that is equivalent to administrator in Windows. Access to all the files and services on that server has now been granted. The worm requests that a copy of itself be transferred to the newly compromised host and starts the process all over again.

Earlier I mentioned how a buffer could be placed on two different kinds of memory – a stack or a heap. Let's look at those now. A stack is a linear list

whose elements are added or removed only in a last-in-first-out (LIFO) order. It is similar in this respect to a stack of books, each book representing one of the contiguous blocks of memory in a buffer. You get to the third book down by first pulling off the top two books. One end of the stack is anchored at a particular address in memory. The other end grows down as information is added to the stack (so, the example of a stack of books is better viewed as an upside-down stack of books sans gravity). A pointer is employed to keep track of the top of the stack, which is the last element added to the stack. Another pointer designates the base of the stack. When an item is removed from the stack (or popped) it isn't actually deleted or erased. The end-of-stack pointer is simply moved back to the previous item. If another item were to be added (or pushed) to the stack at that moment, the previous data would then be overwritten. The location of any item in the stack is determined by counting the number of blocks from the pointer at the base of the stack.

For a definition of heap I quote from the website of an anonymous Irish guy whose site is named A Heap:

> A heap is a space for storing arbitrary quantitys [sic] of various objects. It needs to be able to change size dynamically to suit its contents as they are added and removed on demand. It requires tight management of the resources to be effective and efficient despite often having a haphazard disorganized appearance (Guy, p.1).

It also consists of contiguous memory and is usually located such that it grows up towards the stack. Heaps are dynamically allocated while stacks are more static and predictable. You will hear a lot more about stack overflows because they are much easier to exploit. Aside from that it's useful to know they both exist, as we'll see later in the paper.

The Morris worm is an example of a Stack-Smashing attack – one of the most popular of the buffer overflow exploits. This kind of attack has two goals to accomplish -- you want to deliver the egg into memory and then redirect program control so that the egg is executed (Gillette, p.62). The Morris worm accomplished both objectives in one eloquent package that delivered both the egg and the address of the egg to the original buffer. In other instances the attacker will deliver the egg to another available location in one operation and then overwrite the return address pointer with the egg's location in another attack. The attack isn't considered complete until both objectives have been completed. With the stack only growing in one direction (down), the attacker must locate a block in the buffer that is at a higher address than the address of the return address pointer so that the overflow data can hit its mark. In this context a heap-smashing attack is harder to execute not because of the egg delivery, but in overwriting the program execution path in a dynamic environment.

The success of buffer overflow exploits can be attributed in part to the ready availability of source code for Unix. Overflowing a buffer to have your own code executed is a challenging task. In C there are functions that don't check for bounds before writing. In the Morris worm attack, the *gets ()* function delivered the egg to the buffer. *Gets ()* is notorious in this regard, as are a number of string operations in the standard C library. A potential attacker would save a lot of time by scanning the source code for weak spots in the programming that featured these functions. Fine-tuning a buffer overrun exploit is accomplished more accurately when you can run the program and study the source code. For a long time Windows was thought to be more secure because it didn't ship with its source code. There weren't many people who understood Windows API calls enough to mount a successful buffer overflow in Windows. Prodded by articles such as DilDog's "Tao of Windows Buffer Overflows", attackers have successfully exploited many Windows buffer overflow vulnerabilities. One spectacular success was the Code Red worm, which took advantage of a buffer overflow vulnerability in Microsoft's IIS server in 2001. Buffer overflows in Windows software have been uncovered in increasing numbers ever since.

The buffer overflow is useful to the attacker on several levels. To create a denial of service, the buffer overflow would only have to cripple the program enough to cause it to crash – no egg to deliver, no information to glean, just toast the code. Eliminating a program or server in this way – even if only for a few minutes – could pave the way for another kind of attack. As we saw in the Morris worm, the buffer overflow can also deliver an egg and give the attacker a degree of control on the system. This is achieved usually by compromising a user account on the system. And it can be made to work either locally or remotely. The attacker could also use a buffer overflow to gain elevated privileges on the system, again either locally or remotely. Although a user normally has restricted access to a system, there are sub-routines that temporarily allow a program super-user (or root) privileges in order to accomplish certain tasks, like changing a password. If the attacker can overflow the buffer on those set-user-id (or suid) sub-routines, for example, user privileges can be elevated.

Buffer overflow exploits continue to be popular with would-be attackers. One reason is their seeming omnipresence. Virtually any program that handles data is going to be susceptible. Developers of large retail/commercial software products don't have enough Henry Bakers to check every line of code and track every interaction between routines. The tradeoff between security and profitability ensures a future supply of vulnerabilities. Creating the "egg" (the code that an attacker would write to the buffer) has been a big hurdle in the past. Now there any number of ready-made eggs available. Attackers are thus able to run their code of choice. As network perimeters become increasingly difficult to breach, the buffer overflow attack is able to walk in through the front door. Somewhere behind the firewalls and proxy servers and all the other perimeter devices, there is an application requesting information from the outside world that the attacker is willing to supply in spades. And Y2K did succeed in one aspect

by illuminating to the world the vastness of the sea of legacy software, and in showing how intertwined it is in our daily commerce. Unchecked buffers were being written long before they were being exploited.

Steps to Safeguard Applications

Having endeavored to point out some of the serious security implications posed by buffer overflows, we now look at some of the safeguards available.

An obvious first place to look is bounds checking. If programmers didn't write bounds checking into their code, can't they go back and put it in now? The consensus seems to be that we can and ought to try, but that we'll never get it to 100%. There are many stories about well-used and carefully audited programs that have new vulnerabilities uncovered years later. Much like the attacker who will study source code for signs of weakness, programmers can review code for weak library functions, missing bounds checking, failures to grant the least privileges required and other poor programming practices. Even though the odds seem stacked (after all, the attacker only has to find one vulnerability to be successful), we become incrementally less vulnerable as each programming weakness is resolved. Improved versions of standard C library functions have been developed, but even "safe" functions (like using strncpy instead of strcpy) can fail to terminate a string or can cause other subtle problems if you're not being careful to use them properly.

Tools have been developed to assist in the static checking of code (see Tools at the end of the paper). They vary in complexity, but essentially they take a pre-determined set of parameters and apply them to the code, reducing a program that may be thousands of lines long to a list of hundreds of potential problem areas that are then manually reviewed. Other tools try to watch the program while it is running. Seeing how the variables interact with each other can help predict buffer overflow conditions.

StackGuard (http://www.immunix.org/stackguard.html), a compiler, is a run-time solution that detects stack buffer overflow attacks and stops the victim program before the attack can cause any harm. Though not the same as bounds checking, which would prevent the overflow, StackGuard claims that its performance hit is less than what you would get with bounds checking. It prevents the return address pointer from being overwritten by placing a marker, or canary, just before the return address pointer when the function is called. If the marker has been changed when the function returns, it assumes that something has attempted to alter the address. Again, the program could be helpful. Notice that it doesn't guard against heap overflows (aren't you glad we had that stack/heap discussion?), and that while it may stymie an attack, the end result is the same as a denial of service attack – the program is shut down.

Libsafe (http://www.research.avayalabs.com/project/libsafe/), another run-time solution from Avaya labs, dynamically substitutes certain vulnerable functions from the standard C library with safer functions from its own library that determine the size of the buffer so that it can't be overrun. Although Libsafe is limited in the number of functions it will protect, it won't automatically stop the program when a buffer overflow situation is encountered. It has its own limitations that prevent it from being universally adopted, so be sure to test it thoroughly before production.

Another possible defense is to mark the stack non-executable or stack execute invalidate. In our Morris worm example, the "egg" in the buffer was able to execute. Had the stack been non-executable, that portion of the Morris worm wouldn't have been successful. One drawback is that marking the stack non-executable could break some legitimate functions. It also doesn't prevent buffer overflows. It simply hinders one aspect of them. The attacker could redirect control to a non-stack location where a non-executable stack wouldn't have any influence. The executable "egg" could still run in the heap or in a shared library. In fact, the egg may be nothing more than utilizing a function that already exists in one of the C libraries. Limiting the number of library functions used in the program would limit the number of options available to the attacker and add another layer of defense to the software.

With all the attention being focused on the stack, some programmers have tried writing to the heap in run-time instead of writing data to buffers on the stack. To some degree their programs might be safer because of the fewer heap overflow attacks launched. But they shouldn't consider the buffer overflow problem solved. As stack security increases, more attention may be shifted to exploiting the heap.

For off-the-shelf software the safeguards are more obvious. Check often with the vendor of your operating system for patches. If you are running a vendor-supplied application, again, apply patches as they are made available and isolate the software to minimize the fallout from a compromised system.

Now that we've done what we can on the program side, what if we take a closer look at the data being input into the program? To quote Mr. Grossman, "Again, NEVER TRUST CLIENT-SIDE DATA. Escape, validate, parse, filter and sanity check all the data. With client-side data you can never be too paranoid." (Grossman, slide 32). If the finger server is expecting a user ID in client requests, the programmer should be able to check more than just the number of bytes in the request. Of course, in the days of Y2K we thought that programmers should be able to go into legacy code and correct the handling of dates – no problem. For some applications there may be so many unbounded arrays and poorly written function calls and so few programmers that it may be more practical to clean the data before it hits the program than to try to check it in the program.

Such a data cleaner/checker is called an application firewall. The regular firewalls most familiar to us are the network or perimeter firewalls. They filter packets coming into and leaving the network based on the network layer (OSI Model) information found in the header of each packet. The more specialized application firewall sits between the perimeter firewall and the application host and escapes, validates, parses, filters, and sanity checks the packets for only those protocols and data that apply to the application. It operates at the application layer (OSI Model) and looks beyond the header to allow only the data required by the application. If the application is requesting a first name, the application firewall sees to it that the application receives a string of alpha characters of suitable length and only a string of alpha characters based on rules setup in the firewall. Packets can be reassembled and examined for protocol anomalies and ambiguities. It can also stop potential attackers from being able to probe the application or its host for useful information. While such a solution doesn't entirely compensate for poor programming, it can provide excellent protection from external threats. Be warned that buffer overflow vulnerabilities could still be attacked from within the company through channels not monitored by the application firewall.

AppShield (http://www.sanctuminc.com/solutions/appshield/index.html) is a self-described web application firewall that could fit the bill. Installable either as a gateway or on the host, it provides web intrusion prevention to detect and prevent at least 10 application-hacking attempts before they happen, including buffer overflows. Utilizing its unique policy recognition engine, AppShield can dynamically recognize the application security policy on the fly by analyzing the outbound HTML pages. By being policy-based (as opposed to signature-based), it "enforces intended behavior vs. watching for unintended behavior". The secure proxy architecture can detect encrypted attacks and network details can be hidden from outside users. A number of other features are included and it is expensive software, but may be well worth it in the long haul.

Other efforts to mitigate the impact of buffer overflows have met with limited success. Some have worked at splitting the stack in two to separate the data buffers from the control buffers to help prevent the return address pointer from being overwritten. Various other dynamic detection schemes have been researched and proposed to catch the overflow attempts on the fly or to copy the return address pointer to another location for safekeeping. Only a few representative solutions have been highlighted in this paper.

Conclusion

The demise of the buffer overflow vulnerability has often and long been predicted, yet still they remain. They are too available in too many programs to be ignored. Clever and creative people are working on both sides of the fence,

raising the bar with every exploit and defense. Each of the safeguards mentioned in this paper don't fix the problem, but they do provide an additional layer of protection. There is always a trade-off between competing factors of cost, time, efficiency and the perceived benefit of each safeguard. A series of small hurdles can foil more attacks than one big hurdle. Each different kind of pass you make in reviewing the program's code could uncover previously undetected vulnerabilities. If a StackGuard-type compiler doesn't work for your situation, then maybe the non-executable stack will. It adds one more layer of defense. Any one patch you apply may not seem like much, but when coupled with all the efforts being put into perimeter defenses and system hardening and intrusion detection and staff education and all the many layers of defense, you never know when it might be the proverbial plug in the hole that saves the dike.

Endnotes:

Grossman, Jeremiah. "AFITC 2001 - Web Application Security." 2001.
URL: http://www.whitehatsec.com/presentations/AFITC_2001/afitc_2001.ppt (04 Feb 2004)

Baker, Henry. ""Buffer Overflow" security problems." Email. 26 Dec 2001.
URL: http://www.interesting-people.org/archives/interesting-people/200201/msg00044.html (04 Feb 2004)

Wang, Mengzhi. "15-213 Recitation 4 – 2/18/01." Carnegie Mellon School of Computer Science. 18 Feb 2001.
URL: ftp://docs.jsanet.com/r04-A-4up.pdf (05 Feb 2004)

Guy, Anonymous Irish. Title page.
URL: http://heap.nologin.net/ (04 Feb 2004)

References:

Boettger, Larry. "The Morris Worm: how it Affected Computer Security and Lessons Learned by it." 24 Dec 2000.
URL: http://www.giac.org/practical/gsec/Larry_Boettger_GSEC.pdf (04 Feb 2004)

Dildog. "The Tao of Windows Buffer Overflows."
URL: http://www.cultdeadcow.com/cDc_files/cDc-351/ (04 Feb 2004)

Gillette, Terry Bruce. "A Unique Examination of the Buffer Overflow Condition." Masters Thesis, Florida Institute of Technology. May 2002
URL: http://www.cs.fit.edu/~tr/cs-2002-12.pdf (04 Feb 2004)

McGraw, Gary and Viega, John. "Get reacquainted with the single biggest threat to software security." IBM developerWorks. March 2000
URL: http://www-106.ibm.com/developerworks/library/s-overflows/ (03 Feb 2004)

McGraw, Gary and Viega, John. "Protect your code through defensive programming." IBM developerWorks. Mar 2000.
URL: http://www-106.ibm.com/developerworks/library/s-buffer-defend.html#resources (04 Feb 2004)

Larochelle, David and Evans, David. "Statically Detecting Likely Buffer Overflow Vulnerabilities." 2001 USENIX Security Symposium. Aug 2001.
URL: http://www.cs.virginia.edu/~evans/usenix01-abstract.html (04 Feb 2004)

One, Aleph. "Smashing the Stack for Fun and Profit." Phrack Magazine, Issue 49. 08 Nov 1996.
URL: http://destroy.net/machines/security/P49-14-Aleph-One (04 Feb 2004)

Permeh, Ryan. "The Use of Application Specific Security Measures in a Modern Computing Environment." eEye Digital Security. 22 Mar 2001.
URL: http://www.eeye.com/html/Research/Papers/DS20010322.html (04 Feb 2004)

Thomas, Evan. "Attack Class: Buffer Overflows." Hello World. Apr 1999.
URL: http://www.cosc.brocku.ca/~cspress/HelloWorld/1999/04-apr/attack_class.html (04 Feb 2004)

Tools:

Splint (http://splint.org/)
RATS (http://www.securesoftware.com/)
Flawfinder (http://www.dwheeler.com/flawfinder/)