Interested in learning more about security?

SANS Institute
InfoSec Reading Room

This paper is from the SANS Institute Reading Room site. Reposting is not permitted without express written permission.

Inside the Buffer Overflow Attack:Mechanism, Method, & Prevention

The objective of this study is to take one inside the buffer overflow attack and bridge the gap between the "descriptive account" and the "technically intensive account". The intent is to provide a logical, detailed, and technical explanation of the problem and the exploit that can be well understood by all, including those with little background in the mechanics and methodology of applications programming. We will begin by looking at the "problem" and the problem "mechanism", and then investigate the "means" and the "...

Copyright SANS Institute
Author Retains Full Rights

AD
nitrosecurity Log Management

INSIDE THE BUFFER OVERFLOW ATTACK: MECHANISM, METHOD, & PREVENTION

Mark E. Donaldson
April 3, 2002
GSEC Version 1.3

ABSTRACT

The objective of this study is to take one inside the buffer overflow attack and bridge the gap between the "descriptive account" and the "technically intensive account". The intent is to provide a logical, detailed, and technical explanation of the problem and the exploit that can be well understood by all, including those with little background in the mechanics and methodology of applications programming.

We will begin by looking at the "problem" and the problem "mechanism", and then investigate the "means" and the "method". Based on what we find, we will conclude with recommendations, and a menu for "prevention". Hopefully this approach may also help bridge the gap between "knowledge" and "understanding". Although it may never be possible to purge the world of this security concern, it is certainly within the realm of possibility that the buffer overflow attack be reduced to a level of insignificance through true understanding. Robert Louis Stevenson once wrote: "*Look at yourself. Could you be a doctor, a healing man, with the things those eyes have seen? There's a lot of knowledge in those eyes, but no understanding.*[1]" The technology community must move from fighting buffer overrun attacks defensively to fighting them offensively. To do this, they must transform their knowledge into understanding.

THE PROBLEM

Problematic buffer overruns[2] related to the C programming language data integrity model were first recognized as early as 1973. The first well known exploit of this vulnerability occurred in 1988 when the well documented and infamous Internet Worn shutdown over 6,000 systems in just a few short hours, utilizing an unchecked buffer initialized by the gets() function call in the *fingerd* daemon process. Despite this lengthy history and simple preventative methods, the buffer overflow continues to be a significant and prominent computer security concern even today. For example, buffer overflow problems are implicated in

[1] Robert Louis Stevenson from "The Body Snatcher" published in 1881.

[2] Buffer overflows have assumed several different names over the years. These include buffer overrun, stack overrun, and stack overflow. In practice, all these terms share the same definition and can be used synonymously and interchangeably. Additionally, the stack buffer overflow exploit is often referred to as "stack smashing" in modern day parlance.

© SANS Institute 2002, As part of the Information Security Reading Room. Author retains full rights.

five of the Sans "Top 20" vulnerabilities.[3] If one ventured to the SuSE Linux Web Site, they would find 22 buffer overflow vulnerabilities since January 2001 that require patching (see Table I). Additionally, of the 44 CERT advisories published between 1997 and 1999, 24 were related to buffer overrun issues.

TABLE I
BUFFER OVERFLOW VULNERABILITIES SUSE LINUX

Date	Vulnerability
12.03.2002	buffer overflow in zlib library
04.03.2002	buffer overflow in squid
28.02.2002	revised: buffer overflow in cupsd
28.02.2002	buffer overflows in mod_php4 and mod_php
25.02.2002	buffer overflow in cupsd
25.01.2002	buffer overflow in rsync buffer overflow in rsync
16.01.2002	buffer overflow corruption in /usr/bin/at
07.01.2002	buffer overflow in mutt
24.12.2001	buffer overflow in glibc globbing functions
03.12.2001	buffer overflow problems in openssh
28.11.2001	buffer overflows in wuftpd
10.10.2001	overflow in lpd/lprold
20.09.2001	buffer overflow in WindowMaker
03.09.2001	daemon buffer overflow (nkitb)
23.08.2001	signedness buffer overflow in sendmail
17.08.2001	fetchmail buffer overflow
24.07.2001	(xli) buffer overflow, local+remote
18.04.2001	exploitable buffer overflow in sudo
09.04.2001	xntpd remotely exploitable buffer overflow
27.03.2001	buffer overflow in eperl
22.03.2001	one-byte-buffer overflow in bsd-ftpd and in timed
31.01.2001	buffer overflow in bind8 (new problem; January 2001)

Resident and lingering buffer overruns left in program code are often attributed to "lazy", "sloppy", or "uncaring" programmers, or to modern compilers that fail to perform integrity or bounds checking on their source input or machine output instructions. However, these views may be a bit too simplistic. The root of the problem may run deeper than that. For instance, despite their prevalence today, the buffer overflow vulnerability and attack remain only loosely and informally documented in the literature. From this one might conclude the problem is generally not well understood. This may explain why overrun vulnerabilities continue to appear in new software applications.

Some clarification is necessary here. Indeed, the buffer overflow and exploit problem are well known. Unfortunately, "well known" and "well understood" are often two entirely different views of the same thing. For instance, nearly every book, article, or white paper worth its salt that focuses on computer security

[3] These include W2-ISAPI Extension Buffer Overflows, U1-Buffer Overflows in RPC Services, U3-Bind Weaknesses, U5-LPD (remote print protocol daemon), and U6–sadmind and mountd.

© SANS Institute 2002, As part of the Information Security Reading Room. Author retains full rights.

mentions the buffer overflow vulnerability and their enabling factors. They normally even site preventions or defenses against them. However, they typically avoid discussing or describing the intricate details or complex mechanisms of their cause and their manipulative use in terms that can easily be understood by the novice or inexperienced programmer, system administrator, or computer security practitioner or principal.

Certainly, several detailed accounts of the buffer overflow exploit have been written. These were cited by Nicole LaRock Decker in her GSEC paper "Buffer Overflows: Why, How and Prevention". However, these accounts were written by exceptionally brilliant, and perhaps devious, programmers that chose to jump straight into the details of the low level machine and assembler code necessary to effect the overrun exploit. A typical reader of these accounts usually becomes overwhelmed by the third paragraph, and lays the document to rest.

THE MECHANISM

Buffer overflow vulnerabilities are often attributed to the combined effects of the permissions security features of the UNIX operating system and defects in the C programming language. Such was the premise assumed by Nathan Smith in his 1997 study "Stack Smashing Vulnerabilities In The UNIX Operating System." However, the vulnerability is not just limited to C or UNIX. Indeed, both DilDog and David Litchfield demonstrated the exploit could be used effectively against the Windows NT kernel in their respective papers "The Tao Of Windows Buffer Overflow" and "Exploiting Windows NT4 Buffer Overruns." The paper "Windows NT Buffer Overflow's Start to Finish" shows the problem present and able within the MFC[4] as well. Hence, this paper views the buffer overflow as a language and an operating system independent problem.

Microsoft Corporation defines the buffer overflow attack as follows:

> A buffer overflow attack is an attack in which a malicious user exploits an unchecked buffer in a program and overwrites the program code with their-own data. If the program code is overwritten with new executable code, the effect is to change the program's operation as dictated by the attacker. If overwritten with other data, the likely effect is to cause the program to crash.

Since Microsoft has produced their fair share of buffer overflow vulnerabilities over the years, they should be well versed on the problem and we should not doubt the validity of this description. Thus, we will use this as our official working definition. But what exactly does this mean? And, how do we get there?

[4] MFC is the acronym for Microsoft Foundation Class, Microsoft's C++ OOP library.

© SANS Institute 2002, As part of the Information Security Reading Room. Author retains full rights.

To answer these questions adequately, we must begin by looking at the high level language code and the resultant machine code at the most basic level of the "hardware chain", and investigate how the 80386 processor architecture manages and utilizes memory.[5]

The Buffer Overflow

A buffer overflow is very much like pouring ten ounces of water in a glass designed to hold eight ounces. Obviously, when this happens, the water overflows the rim of the glass, spilling out somewhere and creating a mess. Here, the glass represents the buffer[6] and the water represents application or user data. Let's look a simple C/C++ code snippet that overruns a buffer.

In this function we have a buffer capable of holding eight ASCII characters. Assuming we are on a 32-bit system, this means 16 bytes of memory have been allocated to the buffer. We then place the buffer in an initialization loop and force-feed 15 "x" characters into it through programming error. Obviously they are not all going to fit, and nine of them must spill over into some other memory area like the water overflows its glass. Notice there is no code in this function to check the bounds of the array or to prevent this programming error from occurring. Under most conditions, the overrun of a buffer does not present a security problem in itself. Typically, a segmentation fault will occur and the program will terminate with a core dump. The buffer overflow itself really is that simple. As we shall soon see though, identifying and exploiting the vulnerability complicates matters very quickly.

```
void authenticate ( void )
{
        char buffer1[8];
        int I;

        for ( I = 0; I < 16; I++ )
        {
                buffer1[ I ] = 'x';
        }
}
```

Types of Buffer Overflows

The literature defines the "Stack" and the "Heap" as the two primary types of buffer overrun situations. The stack overflow has two basic variations. One type involves overwriting (and thus changing) security sensitive variables or control

[5] With the exception of VMS, most computer architectures, including the Spark, handle memory use in a similar fashion. One difference that must be considered is the big-endian, little-endian phenomenon. In big-endian architectures, the leftmost bytes (those with a lower address) are most significant. In little-endian architectures, the rightmost bytes are most significant. Many mainframe computers, particularly IBM mainframes, use a big-endian architecture. Most modern computers, including PCs, use the little-endian system. The PowerPC system is *bi-endian* because it can understand both systems.

[6] In its simplest terms, a buffer is a chunk of memory used to temporarily store user data.

© SANS Institute 2002, As part of the Information Security Reading Room. Author retains full rights.

flags stored in memory adjacent to the unchecked buffer. The most common type of stack overflow involves the overwriting of function pointers that can be used to change program flow or gain elevated privileges within the operating system environment. The more complex heap overrun involves dynamic memory allocations, or memory allocated at run time by an application.

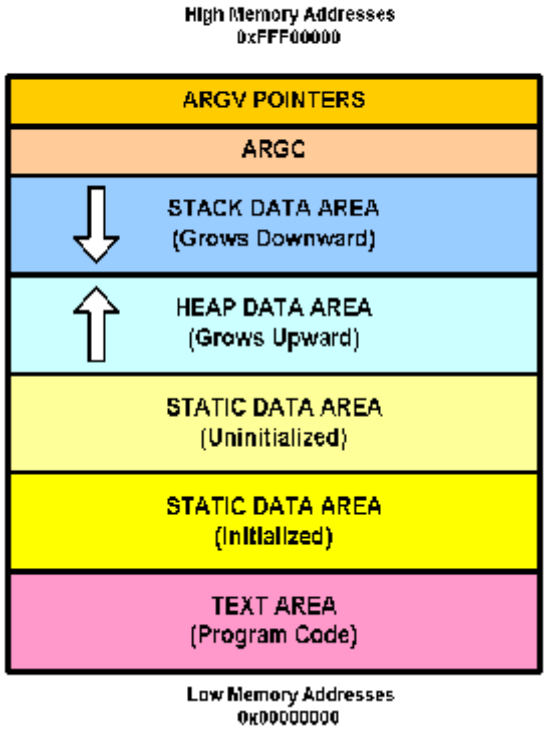
In this study, we will place our focus on the Stack Buffer Overflow. However, in either case, one must have a good understanding of how the operating system allocates memory, and how the application utilizes this allocation. Additionally, this may be the prudent opportunity to define and demonstrate what the stack and heap are and how they work in realistic application.

Structure and Management of Program Memory

Any application or program can logically be divided into the two basic parts of *text* and *data*. Text is the actual read-only program code in machine-readable format, and data is the information that the text operates on as it executes instructions. Text data resides in the lower areas of a processes memory allocation. Several instances of the same program can share this memory area.

Data, in turn, can be divided into the three logical parts of *static*, *stack*, and *heap* data. The distinction between these types is dependant on when and how the memory is allocated, and where it is stored or located. When an executable is first loaded by the operating system, the text segment is loaded into memory first. The data segments then follow. Figure 1 demonstrates these relationships.

Figure 1
The Structure of Program Memory



Static data, located above and adjacent to the text data, is pre-known information whose storage space is compiled into the program. This memory area is normally reserved for global variables and static C++ class members. Static data can be in either an initialized or uninitialized state. Heap data, located above and adjacent to static data, is allocated at runtime by the C language functions *malloc()* and *calloc(),* and by the C++ *new* operator. The heap grows up from a lower memory address to a higher memory address.

© SANS Institute 2002, As part of the Information Security Reading Room. Author retains full rights.

The stack is an actual data structure in memory, accessed in LIFO (last-in, first-out) order. This memory segment, located above and adjacent to heap data, grows down from a higher memory address to a lower memory address. Like heap data, stack data is also allocated at runtime. The stack is like a "scratch pad" that temporarily holds a function's parameters and local variables, as well as the return address for the next instruction to be executed. This return address is of prime importance as it represents executable code sitting on the stack waiting for its turn to execute.

A thorough understanding of the stack and how it functions and performs is essential to understanding how buffer overflow vulnerabilities can be used and exploited for devious and malicious purposes. This being the case, we need to explore the stack and the stack segment in a little more detail. To do this, we will take a temporary and adventurous detour down to the hardware level and into the bowels of the Intel 80386 CPU. Let's begin with the CPU Registers.

Registers

Registers are either 16 or 32 bit[7] high-speed storage locations directly inside the CPU, designed for high-speed access. For the purposes of discussion, registers can be grouped into the four categories of Data, Segment, Index, and Control (see Table II). Certainly, there are some terms here that should seem somewhat familiar. The complete register set is illustrated in Figure 2.

TABLE II
REGISTER SET INTEL 80386 ARCHITECTURE

Category	Register	Function
Data	EAX (accumulator) EBX (base) ECX (counter) EDX (data)	Used for arithmetic and data movement. Each register can be addressed as either a 16 or 32 bit value. EBX can hold the address of a procedure or variable.
Segment	CS (code segment) DS (data segment) SS (stack segment) ES (extra segment) FS & GS	Used as base locations for program instructions, data, and the stack. All references to memory involve a segment register used as a base location.
Index	EBP (base pointer) ESP (stack pointer) ESI (source index) EDI (destination index)	Contain the offsets of data and instructions. The term offset refers to the distance of a variable or instruction from its base segment. The stack pointer contains the offset of the top of the stack
Control	EIP (instruction pointer) EFLAGS	The instruction pointer always contains the offset of the next instruction to be executed within the current code segment.

For instance, the segment registers CS, DS, ES, and SS used as base locations for program instructions (text data), data (static and heap data), and the stack

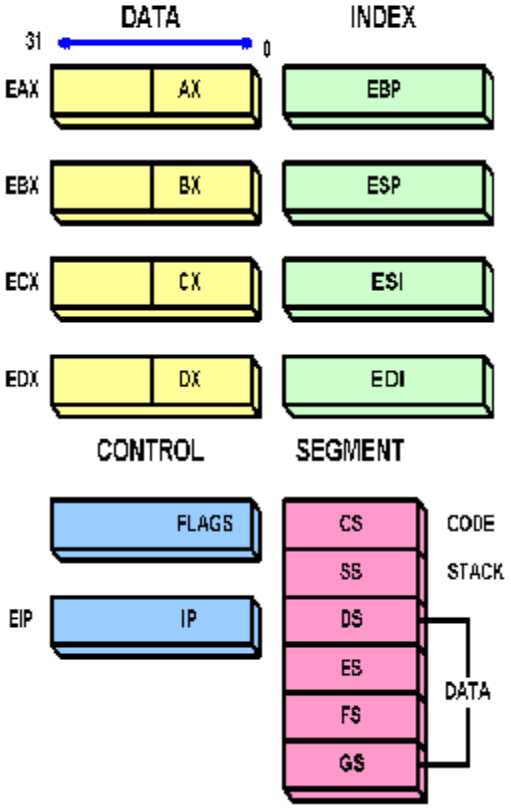
[7] 32-bit and 64-bit in 64 bits systems such as the Alpha and the new Intel Itanium.

© SANS Institute 2002, As part of the Information Security Reading Room. Author retains full rights.

(stack data). The index registers EBP and ESP contain offset references to the code, data, and stack registers. They are, in effect, a compass or positioning service that allow the program to keep track of exactly where all of its data and instructions are located.

The data registers contain actual data bits and are used for the movement and manipulation of this data. EBX is particularly useful for holding the address of a function or variable. EBX plays a crucial role in the exploitation of a buffer overrun. The control registers are bit-wise storage units used to alert the program or CPU of critical states or conditions, within the data or the program itself. EIP is of special importance in that it contains the address of the next instruction to be executed. Again, this is a crucial element in the exploitation of the buffer overrun.

FIGURE 2
THE 80386 32 BIT REGISTER SET



The Stack

Our primary interest, of course, is the stack. Let's look at this data structure a bit closer and see how it relates to and interfaces with the registers. We are forced to look at a little assembly language code at this point, but as you shall see, it is really not all that frightening.

As mentioned earlier, the stack is a special memory buffer outside of the CPU used as a temporary holding area for addresses and data. The stack resides inside of the stack segment. Each 16-bit location on the stack is pointed at by the ESP register, or stack pointer. The stack pointer, in turn, holds the address of the last data element to be added to, or *pushed* onto the stack.

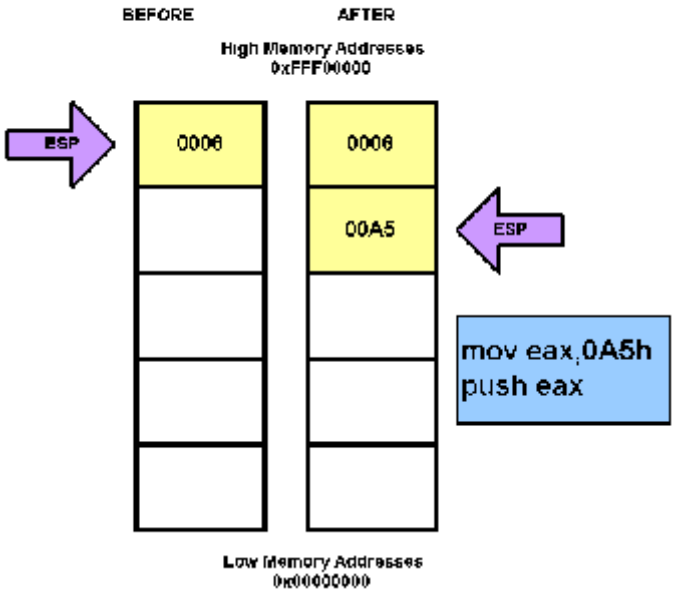
It is important to note that the push operation pushes data backwards onto the stack. This is what causes stack memory to grow downward, or grow toward the lower memory addresses. Now this can truly be confusing and make one's head spin, but it must be understood to execute an attack on the stack. So please, just hang tight.

Conversely, the last value added to the stack is also the first one to be removed, or *popped* from the stack. Hence, the stack is a LIFO (first-in, last-out) data structure. For clarity, let's illustrate this.

© SANS Institute 2002, As part of the Information Security Reading Room. Author retains full rights.

A push operation copies a value onto the stack. When a new value is pushed, ESP (the stack pointer) is decremented. ESP always points to the last value pushed. The *PUSH* instruction is used to accomplish this (Figure 3). The PUSH instruction does not change the contents of EAX, but rather it copies the contents of EAX onto the stack. As more values are pushed, the stack continues to grow downward in memory (Figure 4).

Figure 3
The PUSH Operation



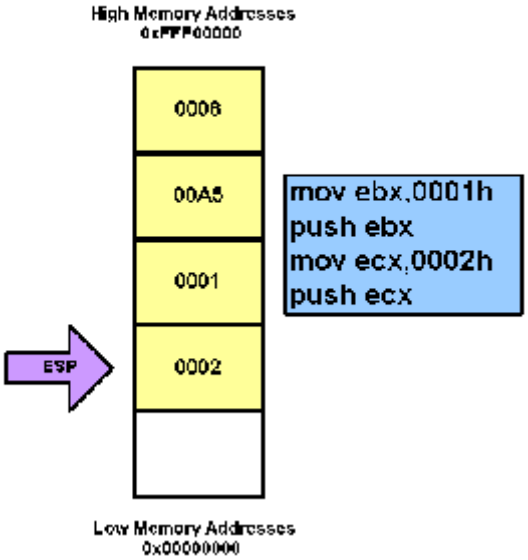
A pop operation removes a value from the top of the stack and places it in a register or variable. After the value is popped from the stack, the stack pointer is incremented to point to the previous value on the stack. The *POP* instruction is used to accomplish this (Figure 5).

Now that we have seen how the stack works at the assembler and machine code level, let's examine this same process from above using C/C++ code.

Stack Operations With C/C++

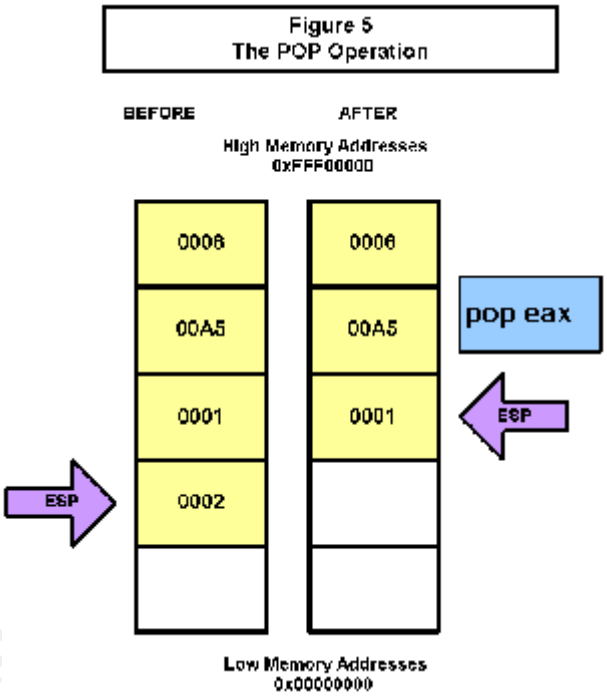
We have now seen how memory is allocated when an executable is first loaded. We have also looked specifically at the stack segment, how a program pushes and pops runtime data to it and from it, and how the stack pointer (ESP) holds the address of the last data element added to the top of the stack. This is all well and good, but it has provided little insight into how buffer overruns are used and exploited. That being the case, it is now time to roll up our sleeves and get down to some serious business. Time is valuable, so lets not mess around any longer. Needless to say, though, this is where things start to get a bit messy and complicated.

Figure 4
The PUSH Operation



© SANS Institute 2002, As part of the Information Security Reading Room. Author retains full rights.

Enter the stack frame pointer (SFP)[8]. The frame pointer always points to a fixed location within the stack frame. Technically speaking, any parameters or local variables that are pushed onto the stack could be referenced by their offsets from the stack pointer (ESP). However, due to the dynamic nature of the stack, these offsets are provided their own reference and are normally stored in the EBP (base pointer) register. In the CPU, this is accomplished by assembly instructions involving both the SFP and EBP. Consequently, function parameters pushed onto the stack will have positive offsets from SFP, while local variables will have negative offsets from SFP[9].



When a function is invoked in the C/C++ language, variables already on the stack must be saved, and space for any new variables must be allocated. The opposite is true when a function exits. When this happens, the prior SFP is pushed onto the stack and a new is SFP is created. ESP then operates with reference to the new local variables. Let's illustrate with some actual code to limit the mass confusion that must be setting in about now.

The following function fragment is drawn from a program developed specially for this study to demonstrate "stack smashing" and the hacking capabilities presented by the buffer overflow situation. It's amazing the damage just a few lines of code can do. But we are not quite ready to unveil it all as we still have a few things to learn first. At this moment, we are only concerned with the sequential order compiled C/C++ code pushes data onto the stack. From this, we may glean some ideas for potential exploits. So let's take a look.

[8] The stack frame pointer (SFP) is sometimes referred to as the local base pointer (BP).

[9] Remember it was stated earlier that stack operations were enough to make your head spin.

© SANS Institute 2002, As part of the Information Security Reading Room. Author retains full rights.

The authenticate() function (shown below) is anything but hypothetical. It's going to do some real work for us. However, it has been temporarily modified so we can learn from it. Authenticate() is designed to authenticate a user attempting to "login" to a computer system. Access is then either permitted or denied. Authenticate() accepts two string pointers as arguments (passed in as parameters). The two 16 byte parameters consist of a password entered by the user, and a password obtained from the system, presumable either from the SAM database in Windows NT or the /etc/passwd (/etc/shadow/) in UNIX[10].

```
void authenticate ( char * string1, char * string2 )
{
    char buffer1[8];
    char buffer2[8];

    printf ("The address of string2 is %x.\n", &string2 );
    printf ("The address of string1 is %x.\n", &string1 );
    printf ("The address of buffer1 is %x.\n", &buffer1 );
    printf ("The address of buffer2 is %x.\n", &buffer2 );

    strcpy ( buffer1, string1 );
    strcpy ( buffer2, string2 );
}
```

Additionally, authenticate() contains two local 16 byte data buffers[11] into which the password parameter values will copied. Through a little programming trickery[12] we can actually watch the parameter values and local buffers be pushed onto the stack. The compiled assembler code, minus the printf() statements, appears as follows:

```
push $2             ; push authenticate() argument 2
push $1             ; push authenticate() argument 1
call authenticate   ; call authenticate() and push EIP onto the stack
push %ebp           ; push frame pointer onto the stack
mov %esp,%ebp       ; copy ESP into EBP creating new  frame pointer (FP)
sub $16,%esp        ; allocate local variable space by subtracting size from ESP
```

When executed, this code first pushes the two arguments to authenticate() backwards onto the stack. It then calls authenticate(). The instruction *CALL* then pushes the instruction pointer (EIP) onto the stack[13]. Authenticate() is now free

[10] Windows NT implies any Windows system utilizing the core NT kernel to include Windows 2000 and Windows.NET. UNIX includes the feisty UNIX clone, Linux.

[11] It is important to note that even though these buffer arrays consist of eight ASCI characters, each "char" data type is allocated two bytes (one word) of memory in a 32-bit system.

[12] To do this, we will supply the hexadecimal notation (%x) to the C printf() function in place of the string notation (%s).

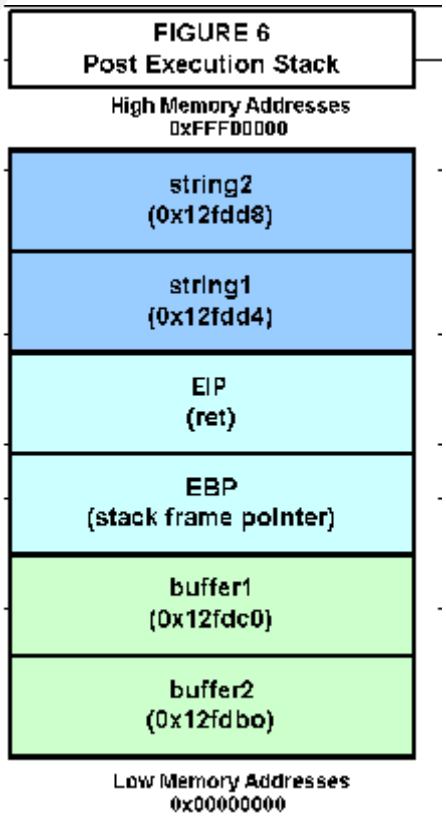
[13] Referred to as ret, EIP is now holding the address of the next code to be executed one the authenticate() function exits.

© SANS Institute 2002, As part of the Information Security Reading Room. Author retains full rights.

to execute on its own. First, it pushes the stack frame pointer onto the stack. The current stack pointer (ESP) is then copied into EBP, making it the new frame pointer (SFP). Next, memory space is allocated to the local buffers by subtracting their size from ESP. Finally, the printf() functions kick-in to verify that indeed this is what is occurring on the stack. Here is what we see when we run this block of code:

```
The address of string2 is 12fdd8.
The address of string1 is 12fdd4.
The address of buffer1 is 12fdc0.
The address of buffer2 is 12fdb0.
```

What we expected would happen, in fact, did happen. First, parameter 2 (char pointer string2) is pushed onto the top of the stack at address 0x12fdd8, thereby assuming the highest memory address. Next, we see parameter 1 (char pointer string1) pushed onto the top of the stack, but backwards from parameter 2. Since the stack grows downward toward lower memory address, we should expect parameter 1 to hold a lower memory address. In fact, it holds address value 0x12fdd4, exactly four bytes down for parameter 2[14].



Next, although not exposed here, instruction pointer (EIP) and the stack frame pointer (EBP) are each pushed in respective order. Finally, memory for the two local buffers is allocated on the stack. With the stack continuing to grow downward, buffer1 takes address 0x12fdc0, and again as expected, buffer2 grabs address 0x12fdb0 at the top of the stack, exactly 16 bytes down from buffer1.

Wow. What a mouthful. To simplify this dribble and put everything in proper perspective, let's diagram the stack as it exits in its present condition (Figure 6). What we have here is a stack just

[14] Since the parameter values were both 16 byte values, you might be wondering why parameter 1 is four bytes down and not 16 bytes down. In C++, parameters may be passed either by value or by reference. Since they were passed by reference in this case, they are actually holding the 32-bit address of the value, and not the value itself. Had they been passed by value, their addresses would be 12feb0 and 12fec0 respectively.

© SANS Institute 2002, As part of the Information Security Reading Room. Author retains full rights.

waiting to be “smashed”. The time has finally come to explore the methods and tactics behind the buffer overflow attack. Let’s put on our black hats and become the attacker.

THE METHOD

For a buffer overrun attack to be possible and be successful, the following events must occur, and in this order:

1. A buffer overflow vulnerability must be found, discovered, or identified.

2. The size of the buffer must be determined.

3. The attacker must be able to control the data written into the buffer.

4. There must be security sensitive variables or executable program instructions stored below the buffer in memory.

5. Targeted executable program instructions must be replaced with other executable instructions.

Let’s look at each of these five conditional steps individually.

Step 1: Discovery and Identification

There are four primary means by which discovery or identification may take place:

1. By the reporting of others, albeit by white hat security alert or bulletin, or through the black hat underground.

2. By scrutinizing source code.

3. By accident or stroke of luck.

4. By brute trial and error, utilizing intentional and systematic means.

The first two means are quite obvious and warrant no discussion here. Accidental discovery may often be unrecognized as such, with the end result being a “crash dump” of the UNIX or NT system, or the proverbial MessageBox informing the Windows user that their program has either performed “an illegal operation and will be shutdown”, or their program “instruction referenced memory that could not be read”. However, the savvy or devious minded user might take notice of the potential significance and investigate further by employing brute trial and error through intentional and systematic means.

© SANS Institute 2002, As part of the Information Security Reading Room. Author retains full rights.

Trial and error tactics might be used from the start by those who have far too much time on their hands. The trail and error process literally involves the repetitive feeding of varying length input into a program or application[15]. By chance, if the "your program has performed an illegal operation and will be shutdown" or "your program instruction referenced memory that could not be read" message arises[16], then "Bingo!! At this point, additionally investigation through yet more trial and error is required to determine if indeed a buffer overrun has been discovered. Once the overrun is confirmed, the real brainwork begins.

The Demonstration Code

This may be a good time to break out the full version of our demonstration program and begin illustrating the exploit procedure. Earlier we were introduced to the authenticate() function for illustrating stack operation. Here is the entire code block for the program authenticate.exe:

```
int main ()
{
    char name[8];
    char etc_passwd[8];
    char password[8];

    // retireive the user information
    printf( "Enter your name: " );
    gets( name );
    etc_passwd = get_password ( name );
    printf( "Enter your password: " );
    gets( password );
    printf( "Your name and password entires were %s and %s.", name, password );
    printf( "The password for %s in the /etc/shadow file is %s.", name, etc_passwd );

    // call procedure to check login authorization
    authenticate ( password, etc_password );
    return 0;
}
void authenticate ( char * string1, char * string2 )
{
    char buffer1[8];
    char buffer2[8];
    strcpy ( buffer1, string1 );
    strcpy ( buffer2, string2 );

    if ( strcmp ( buffer1, buffer2 ) == 0 ) permit();
}
```

[15] This is commonly done utilizing looping functions in scripts, effectively "automating" the process.

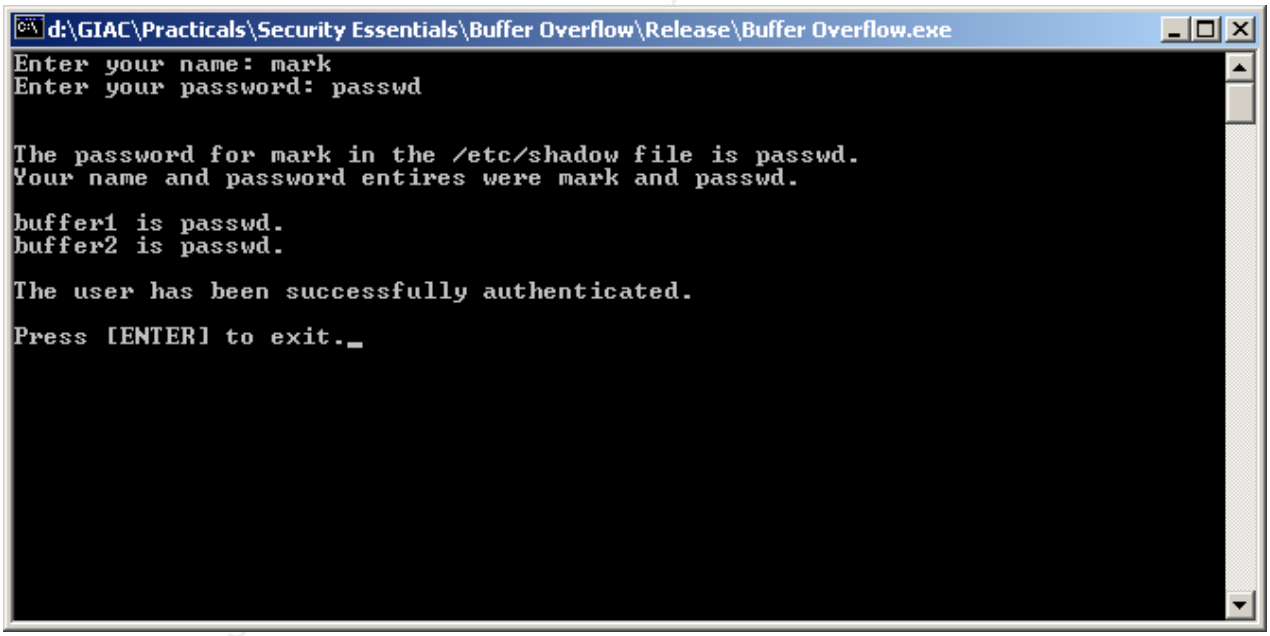
[16] It should be noted that many different types of errors may produce these messages. They are not all indicative of a buffer overrun.

© SANS Institute 2002, As part of the Information Security Reading Room. Author retains full rights.

Authnticate.exe is a simple program, and quite frankly, one exactly like it is not likely to be found in a production environment. Let's hope not anyway. However, it should prove more than adequate to show us how a hacker works on an unchecked buffer. Here is how the program works.

First, main() allocates three array variables to contain the information necessary to authenticate a user logging into to a computer system. The program requests that the user input their name and password. This information is obtained by calling the notoriously flawed[17] C subset gets() function. Once main() has the users "username" in hand, it calls the function get_password() to acquire the expected authenticating "password" from the system database. It then passes the "input password" and the "system password" to our old friend authenticate() for further processing.

Next, authenticate() promptly copies the two password values into their respective buffers using the flawed strcpy() function. Finally, the contents of the two buffers are compared by calling the flawed strcmp() function. If the two values match, the user is authorized, and they are permitted to use the system. Now, let's run it.

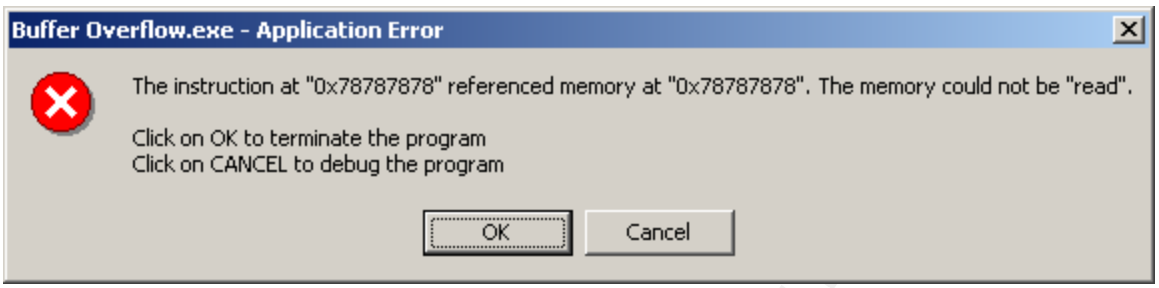


Seems to work pretty good to me. As you can see, the user entered the username of "mark" and the password of "passwd". This matched the entries in the /etc/shadow file, and the user was successfully authenticated. So, what's the problem?

[17] Flawed means it happily and willingly accepts a buffer value as an argument without first checking its size or limits. Notorious means the "flaw" is well known and an easy target for exploit

© SANS Institute 2002, As part of the Information Security Reading Room. Author retains full rights.

To answer this question, let's back track to Step 1 (Discovery and Identification) of the buffer overrun attack, and begin feeding varying length input into the program's password request. And Bingo!! We hit pay dirt.



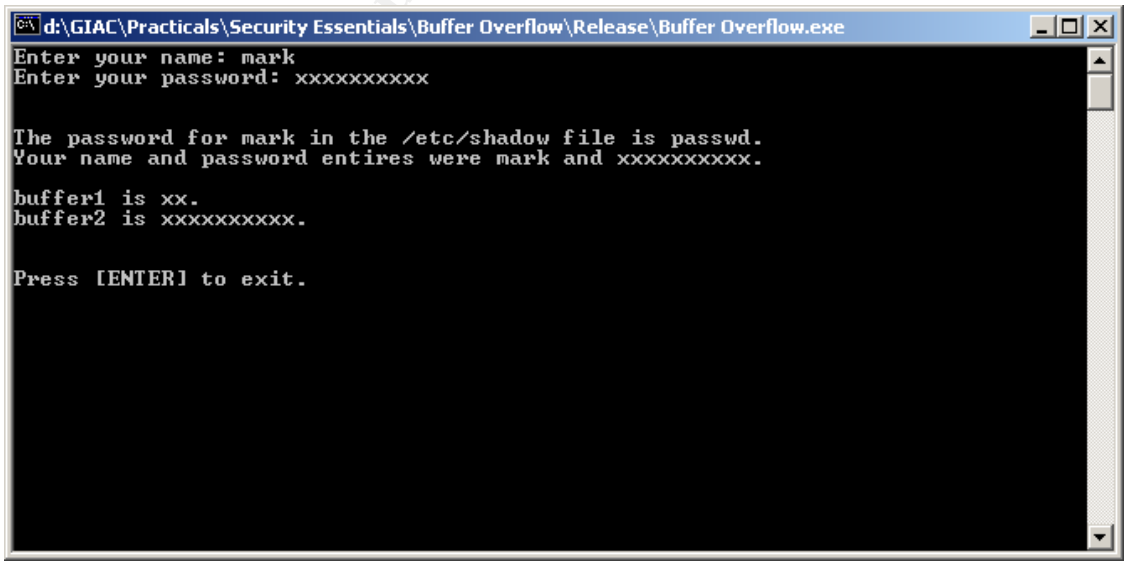
We just overran a buffer and discovered a overflow vulnerability. Now Step 2.

Step 2: Determine Buffer Size

Before we can do much with our newly discovered buffer problem, we need to determine the exact size of the buffer. We can do this through experimentation, and by slowly growing and shrinking the number of characters we input into the buffer. As soon as we determine the exact number of characters it takes to crash the program, we have completed Step 2. In this case, it took little work to figure out this was an eight character array (seven chars plus one null terminator). We now know at which point the buffer begins to overflow. Now Step 3.

Step 3: Control Data Written Into Buffer

Based on what we have learned about this program thus far, this step may be a no brainer. Let's feed in a series of x's and see if we are able to control what happens within the program.

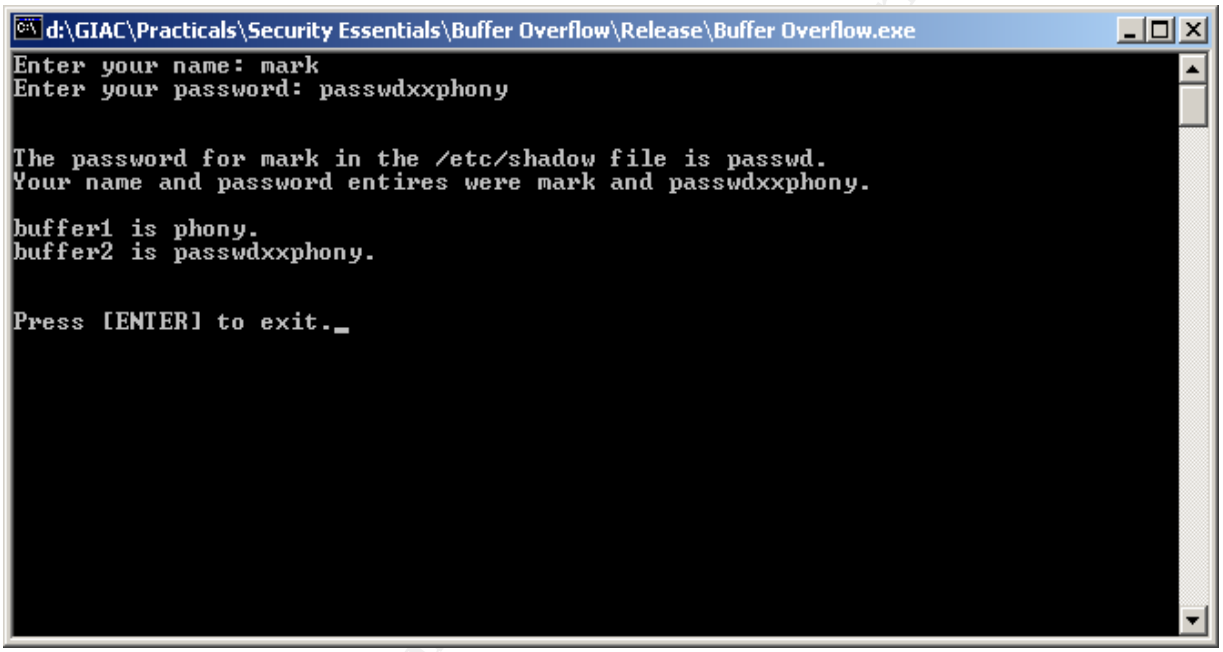


© SANS Institute 2002, As part of the Information Security Reading Room. Author retains full rights.

Seems as though we can. First, we entered the same username of “mark”. As an obedient program would, it successfully retrieved the correct password of “passwd” for user “mark“ from the /etc/shadow file. However, instead of entering the correct password, we entered 10 x’s. Something dreadful has happened to what the program believes to be the “system password”. Step 3 completed.

Step 4: Overwrite Security Sensitive Variables Below The Buffer

There are several possibilities of action once the hacker has reached this stage of the process. In our Step 4, we will attempt to overwrite a security sensitive variable by overflowing the vulnerable buffer with input of our choice, thereby affecting the outcome of the program. Here we go.



Observe that we just changed the “system password” from “passwd” to “phony” by overwriting it with input of our choice. We, the hacker, now have complete control of this program. Depending on what we wish to accomplish, there are several different directions in which we could go. First, let’s take another look at the stack and see exactly what has occurred to this point (Figure 7).

First, the two password parameters (password and etc_password) were passed into authenticate() by reference. When authenticate() executes, strcpy() is called to perform a bit-by-bit copy of etc_password into buffer1. Next, strcpy() is again summands to perform a bit-by-bit copy of password into buffer2. Unfortunately, strcpy() pays no attention to the size or the contents of either password or buffer2. Buffer2 loads up with eight char values, but strcpy() blindly continues to pump the additional values from our input into buffer2. The real problem is they don’t fit, but must go somewhere. Alas, the glass begins to overflow. Due to the

© SANS Institute 2002, As part of the Information Security Reading Room. Author retains full rights.

inherent nature of stack operation, the additional char values literally overflow buffer2 and spill over into buffer1. As an end result, we successfully changed the "password" value that is to be matched against our input for authentication. Thus, we now control both values at will.

Step 5: Replace Targeted Executable Instructions With Other Instructions

Most black hats won't be satisfied with this achievement alone however. They are generally seeking larger and more grandeur things, such as elevating their privilege in the system by becoming "root" or "administrator". That leads us to Step 5.

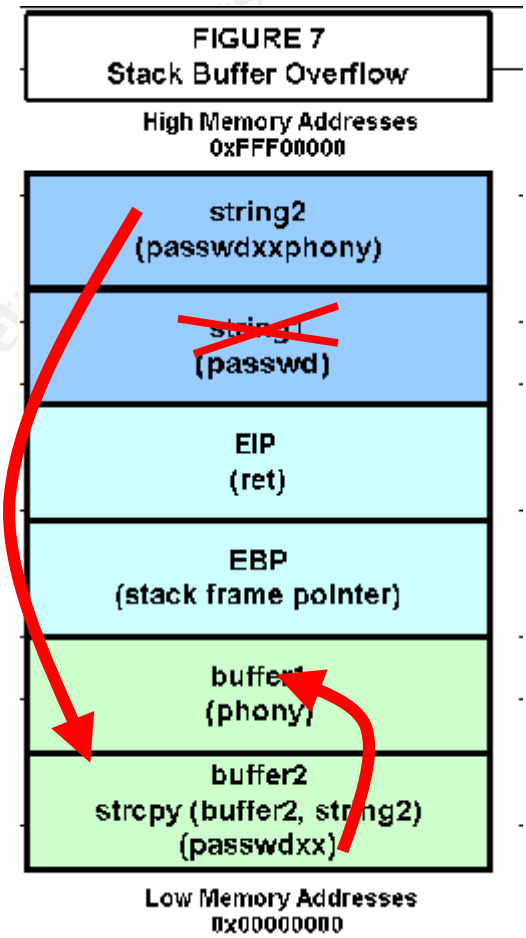


FIGURE 7
Stack Buffer Overflow

We have learned a great deal about the stack up to this point, as well as our demonstration code. For instance, when the *CALL* to authenticate() was made, the program and processor needed to mark its trail so they would know where to go once authenticate() completed its execution. To accomplish this, the instruction pointer (EIP) was pushed onto the stack directly after authenticate()'s parameters. EIP holds this "return address", and this is the address of an executable instruction. We observe that EIP also resides on the stack below the memory buffer we have already proven can be manipulated at will. This indeed presents a tempting situation. Before we proceed, let's first, let's take a quick look at system processes and privileges.

System Processes and Privileges

Although the internal mechanism works quite differently, both Windows NT and UNIX have a commonality in the manner new processes are created. For instance, in UNIX, when a process forks or creates a child process, the child process generally has the same privilege level as its parent process. Consequently, if a program was configured with SUID "root" permissions and made world executable, any process it spawned would most likely have identical privileges. This would include any UNIX shell that might be spawned.

Similarly, in Windows NT, when a process starts a child process, the child process normally inherits the access token[18] of the parent process[19].

[18] In Windows NT, an access token is a numeric key that determines privilege level and what a user can do on that machine.

© SANS Institute 2002, As part of the Information Security Reading Room. Author retains full rights.

Consequently, if a program running with system or administrative privileges were to launch a "command shell", the newly created shell process would normally inherit the access tokens of its parent process. Now that we know this, we can continue with Step 5 of our exploit.

It should now be apparent that our goal, as hacker, is to obtain elevated system privilege using the problematic buffer we have been working with. Since our authentication program is doubtlessly running with system privilege, it may be very possible to achieve this goal utilizing the executable code EIP will direct us to. Since we are working on a Windows machine, we will attempt to execute a Windows command shell "cmd.exe" using the system() function. Here is the game plan[20]:

1. Push EBP onto the stack as relative stack reference pointer.

2. Create and push a NULL on the stack as string "cmd.exe" must have a null terminator. This is done by "xoring" a register with itself (xor edi, edi; push edi).

3. Push the code we want to execute (cmd.exe) onto the stack and use EBP to position and track its starting address. To do this, we will need to push each byte individually and in reverse order (exe.dmc).

4. Push the address of the system() command (0x780208C3) on the stack to overwrite EIP (0x015DF124) (mov eax, 0x780208C3; push eax).

5. Push the starting address of cmd.exe onto the stack with reference from EBP (lea eax,[ebp-08h]; push eax).

6. Call system() with reference to EBP (call dword ptr [ebp-0ch]).

With the stack set up in this fashion, the normal course of program execution will call system(), which in turn will launch "cmd.exe." Goal achieved. Game over[21].

---

[19] There are exceptions. For instance some processes can be started using the Win32 API CreateProcessAsUser() function that will start the new process under the security context of another user. In this case, the new process will have a different access token than the parent process.

[20] The precise code for this part of the exploit has intentionally been withheld as this is an educational presentation, and not a "How To." However, for those inclined, enough detail has been provided to reproduce it with sufficient effort.

[21] This same exploit could be easily be executed on a UNIX box using the execve() function and "/bin/sh", or any other UNIX shell.

© SANS Institute 2002, As part of the Information Security Reading Room. Author retains full rights.

PREVENTION

Although we have just demonstrated a nightmare scenario for any system administrator, we must bear in mind that ALL such buffer overflow attacks are very preventable, and the "disease" that allows them to persist certainly may be eradicated in the future. However, an effective vaccine must first be developed. The remainder of this report will focus on cure and prevention. Please select from the menu.

1. **Use Different Language Tools**. Language tools that provide automatic bounds checking such as Perl, Python, and Java. True, these are available. However, this is usually not possible or practical when you consider almost all modern operating systems in use today are written in the C language. The language tool becomes particularly critical when low-level hardware access is necessary. The good news is with languages evolving, language and code security has becoming a serious issue. For example, Microsoft in their .NET initiative has completely re-written Visual Basic and Visual C++[22] with "string safe" security in mind. Additionally, they have added the Visual C# tool which was designed from the ground up with security in mind.

2. **Eliminate The Use Of Flawed Library Functions**. Programming languages are only as flawed as the programmer allows them to be. In our demonstration, we utilized three flawed functions from the Standard C Library (gets(), strcpy, and strcmp). These are just three of many such functions that fail to check the length or bounds of their arguments. For instance, we could have completely eliminated the buffer overflow vulnerability in our demonstration by changing one line of code. This simple change informs strcpy() that it only has an eight byte destination buffer and that it must discontinue raw copy at eight bytes.

```
// replace line 49 from
strcpy ( buffer2, string2 );

// to
strncpy ( buffer2, string2, 8 )
```

   The persistence of programming errors of this nature may indeed be related to the manner in which we train and educate young programmers. One can pick up an introductory college textbook on C or C++ and find this set of flawed functions introduced by the third chapter. Sure, they make great training aids. However, humans are creatures of habit and tend to use what they know best and are most comfortable with.

3. **Design And Build Security Within Code**. It takes more work, and it takes more effort, but software can be designed with security foremost in mind. If

---

[22] Visual C++ is Microsoft's proprietary version of the C++ language.

© SANS Institute 2002, As part of the Information Security Reading Room. Author retains full rights.

the previous example, we could have yet added one extra step to assure complete buffer safety:

```
strncpy ( buffer2, string2, sizeof( buffer2 ) )
```

Again, this may go back to how we train programmers. Is code security taught and encouraged? Are they given the extra time to design security within their code? Typically, and unfortunately, the answer to these questions is no[23].

4. **Use Safe Library Modules**. String safe library modules are available for use, even in problematic languages such as C++. For instance, the C++ Standard Template Library offers the Class String in its standard namespace. The String Class provides bounds checking within its functions and be preferred for use over the standard string handling functions.

5. **Use Available Middleware Libraries**. Several freeware offerings of "safe libraries" are available for use. For instance, Bell Labs developed the "libsafe[24]" library to guard against unsafe function use. libsafe works on the structure of stack frame linkage through frame pointers by following frame pointer to the stack frame that allocated a buffer. When a function executes, it can then prevent the return address from being overwritten. However, libsafe is not without security problems of its own as it has been reported that libsafe's protections can be bypassed in a format-string-based attack by using flag characters that are used by glibc but not libsafe. Users of libsafe should upgrade to version 2.0-12

6. **Use Source Code Scanning Tools**. Several attempts have been made to design a tool that performs analysis on raw source code with the hope of identifying undesirable constructs to include buffer vulnerabilities. The boys at L0pht Heavy Industries (now a white hat group called @atstake) produced one such tool called "Slint" a few years back, but it was never released. Probably the most successful tool to date is Rational's (http://www.rational.com) PurifyPlus Software Suite that capably performs a dynamic analysis of Java, C, or C++ source code. Although the specialty of PurifyPlus is memory leak detection, it is capable of hunting down unchecked

[23] It was recently reported in the media that Bill Gates went on a tirade about code security at Microsoft In fact, he reportedly stopped all new code production for one month to train his programmers in code security basics. Until now, Microsoft has stressed form and function at the expense of security.

[24] Libsafe has apparently now been turned over to Avaya Labs http://www.avaya.com for keeping and maintenance.

© SANS Institute 2002, As part of the Information Security Reading Room. Author retains full rights.

buffers and other coding errors that could possibly lead to buffer overrun conditions.

7. **Use Compiler Enhancement Tools**. Although a relatively new concept, several compiler add-on tools have recently been made available that work closely with function return address space to prevent overwriting. One such tool, Stack Shield (http://www.angelfire.com/sk/stackshield), provides protection by taking a copy of RET and temporarily placing it in a location not subject to overflow attacks. Upon return, the two address values are compared. If they are different, the return address has been modified and Stack Shield terminates the program. A somewhat similar tool, Stack Guard (http://www.immunix.org/stackguard.html), is able to detect a return address being overwritten in real time. When it does, it proceeds to terminate the program.

8. **Disable Stack Execution**. Although it requires the operating system kernel to be recompiled, patches are available for some versions of UNIX that render the stack non-executable. Since most buffer overrun exploits depend on an executable stack, this modification will essentially stop them dead in their tracks. A patch for the Linux kernel has been made available by the Openwall Project (http://www.openwall.com).

9. **Know What Is On Your System**. Awareness of what is on your system and who has the privileges to execute it is essential. SUID root executable, and root owned world writable files and directories are the favorite target of many attacks. Find them, list them, and know them. The following few simple commands may be your best friend:

```
( echo "SUID-ROOT LIST" ; find / -user root -type f -perm -4000 ) | lpr
( echo "WORLD WRITABLE LIST" ; find / -user root -perm -022 ) | lpr
```

   Once your list is complete and in hand, programs are available to test each for buffer overrun vulnerabilities. Should a “segmentation fault” be occur during testing, chances are you have just discovered a vulnerable program.

10. **Patch The Operating System And Application**. Perhaps the very best defense is to stay informed and remain “offensive”. As new vulnerabilities are discovered and reported, apply the necessary patches and fixes promptly. If you are in a Microsoft shop, this may get very tiresome very quickly. It may even seem like an endless task. But cheer up. Knowledge in increasing and understanding is improving. The diseased will be cured.

© SANS Institute 2002, As part of the Information Security Reading Room. Author retains full rights.

REFERENCES

Aleph One. "Smashing The Stack For Fun And Profit." Phrack Magazine. Issue #49 November 1996. URL: http://destroy.net/machines/security/P49-14-Aleph-One.

Baratloo, Arash et al. "Libsafe: Protecting Critical Elements of Stacks." Bell Labs. December 25, 1999. URL: http://www.avayalabs.com/project/libsafe/doc/libsafe.pdf.

Cowan, Crispin et al. "Protecting Systems From Stack Smashing Attacks With StackGuard." Department of Computer Science and Engineering. Oregon Graduate Institute of Science & Technology. URL:http://www.cse.ogi.edu/DISC/projects/immunix/.

dark spyrit. "Win32 Buffer Overflows." Phrack Magazine. Issue #55 September 1999. URL: http://julianor.tripod.com/P55-15-win32_overflow.txt.

dethy. "How To Write Code Based Exploits." March 2000. URL: http://julianor.tripod.com/htce.txt.

DilDog. "The Tao Of Windows Buffer Overflow." URL: http://www.cultdeadcow.com/cDc_files/cDc-351.

Farrow, Rik. "Blocking Buffer Overflow Attacks." Network Magazine. November 1999. URL: http://www.networkmagazine.com/article/NMG20000511S0015.

Frykholm, Niklas. "Countermeasures Against Buffer Overflow Attacks." RSA Security. November 2000. URL:http://www.rsasecurity.com/rsalabs/technotes/buffer/buffer_overflow.html.

Harari, Eddie. A Look At The Buffer Overflow Hack." Linux Journal. Issue #61 May 1999. URL: http://www.linuxjournal.com/article.php?sid=2902.

Henry, Paul A. "Buffer Overrun Attacks". Cyberguard Corporation. URL: http://www.utdallas.edu/~aph3x/docs/programming/security/overruns.pdf.

Kalev, Danny. "Avoiding Buffer Overflows." ITworld.com Newsletter. December 2001. URL: http://www.itworld.com/nl/lnx_sec/12182001/pf_index.html.

Lamagra. "Buffer Overflows." URL: http://julianor.tripod.com/lamagra-bof.txt.

Lefty. "Buffer Overruns: What's The Real Story?" URL: http://julianor.tripod.com/stack-history.txt.

© SANS Institute 2002, As part of the Information Security Reading Room. Author retains full rights.

Litchfield, David. "Exploiting Windows NT4 Buffer Overruns." URL: http://www.atstake.com/research/reports/wprasbuf.html.

Litchfield, David. "Analysis Of The winhlp32.exe Buffer Overrun." URL: http://www.cerberus-infosec.co.uk/wpwhlpbuf.html.

Mazidi, Muhammad Ali and Mazidi, Janice Gillispe. The 80x86 IBM PC and Compatible Computers (Volumes I & II). Upper Saddle River: Prentice Hall, 1998.

Microsoft Security Bulletin MS02-014. "Unchecked Buffer In Windows Shell Could Lead To Code Execution." March 7, 2002. URL: http://www.microsoft.com/technet/treeview/default.asp?url=/technet/security/bulletin/MS02-014.asp.

Mixter. "Writing Buffer Overflow Exploits – A Tutorial For Beginners." URL: http://www.11a.nu/stack/exploit.txt.

Mazidi, Muhammad Ali and Mazidi, Janice Gillispe. The 80x86 IBM PC and Compatible Computers (Volumes I & II). Upper Saddle River: Prentice Hall, 1998.

mudge. "How To Write Buffer Overflows." URL: http://www.insecure.org/stf/mudge_buffer_overflow_tutorial.html.

Smith, Nathan P. "Stack Smashing Vulnerabilities In The UNIX Operating System." 1997. URL: http://www.bronzesoft.org/docs/security/bufov/nate-buffer.pdf.

Sorfa, Petr. "Debugging Memory On Linux." Linux Journal. Issue #87 July 2001. URL: http://www.linuxjournal.com/article.php?sid=4681.

Taeho Oh. "Advanced Buffer Overflow Exploit." 1999. URL: http://online.securityfocus.com/library/1568.

Tarreau, Willy. "Security Under Linux: The Buffer Overflow Problem." URL: http://www-miaif.lip6.fr/willy/security/linux.html.

teleh0r. "Buffer Overflows For Kidz." URL: http://julianor.tripod.com/bof-forkidz.txt.

Tsai, Timothy and Navjot Singh. "Libsafe 2.0: Detection of Format String Vulnerability Exploits." Avaya Labs. February 6, 2001. URL: http://www.avayalabs.com/project/libsafe/doc/whitepaper-20.pdf.

© SANS Institute 2002, As part of the Information Security Reading Room. Author retains full rights.

Unknown. "Stack Overflow Exploits On Linux/BSDOS/FreeBSD/SunOS/Solaris/HP-UX." URL:http://julianor.tripod.com/thc3-en.txt.

Unknown. "Windows NT Buffer Overflow's From Start To Finish." URL:http://www24.brinkster.com/neolabs/papers/bufferoverflows/nt-bofstf.txt.

© SANS Institute 2002, Author retains full rights.

© SANS Institute 2002, As part of the Information Security Reading Room. Author retains full rights.

Last Updated: May 22nd, 2013

Upcoming SANS Training

Click Here for a full list of all Upcoming SANS Events by Location

SANS Malaysia @ MCMC 2013	**Cyberjaya, MY**	**Jun 03, 2013 - Jun 08, 2013**	**Live Event**
SANS Pen Test Berlin 2013	**Berlin, DE**	**Jun 03, 2013 - Jun 08, 2013**	**Live Event**
Industial Control Systems Security Training - Houston	**Houston, TXUS**	**Jun 10, 2013 - Jun 15, 2013**	**Live Event**
Security Impact of IPv6 Summit 2013	**Washington, DCUS**	**Jun 14, 2013 - Jun 16, 2013**	**Live Event**
SANSFIRE 2013	**Washington, DCUS**	**Jun 14, 2013 - Jun 22, 2013**	**Live Event**
SANS Canberra 2013	**Canberra, AU**	**Jul 01, 2013 - Jul 13, 2013**	**Live Event**
Digital Forensics & Incident Response Summit 2013	**Austin, TXUS**	**Jul 09, 2013 - Jul 16, 2013**	**Live Event**
SANS London Summer 2013	**London, GB**	**Jul 09, 2013 - Jul 16, 2013**	**Live Event**
SANS Rocky Mountain 2013	**Denver, COUS**	**Jul 14, 2013 - Jul 20, 2013**	**Live Event**
SANS Mumbai 2013	**Mumbai, IN**	**Jul 22, 2013 - Jul 27, 2013**	**Live Event**
SEC528 SANS Training Program for the CompTIA® New Advanced Security Practitioner (CASP) Certification	**Washington, DCUS**	**Jul 22, 2013 - Jul 26, 2013**	**Live Event**
SANS San Francisco 2013	**San Francisco, CAUS**	**Jul 29, 2013 - Aug 03, 2013**	**Live Event**
SANS SEC 560: Network Penetration Testing @ Bangalore 2013	**Bangalore, IN**	**Aug 05, 2013 - Aug 10, 2013**	**Live Event**
SANS Boston 2013	**Boston, MAUS**	**Aug 05, 2013 - Aug 10, 2013**	**Live Event**
Critical Security Controls Summit	**Washington, DCUS**	**Aug 12, 2013 - Aug 18, 2013**	**Live Event**
Industrial Control Systems Security Training - DC	**Washington, DCUS**	**Aug 12, 2013 - Aug 16, 2013**	**Live Event**
SANS Thailand 2013	**Bangkok, TH**	**Aug 19, 2013 - Aug 31, 2013**	**Live Event**
SANS Virginia Beach 2013	**Virginia Beach, VAUS**	**Aug 19, 2013 - Aug 30, 2013**	**Live Event**
Mobile Device Security Summit 2013	**OnlineCAUS**	**May 30, 2013 - Jun 06, 2013**	**Live Event**
SANS OnDemand	**Books & MP3s OnlyUS**	**Anytime**	**Self Paced**