Chapter 14

Remote identification of the operating system

To carry out an attack successfully requires planning. An important part of this is gathering information about the target. From the hacker's point of view, the most essential piece of knowledge to be obtained is the version of the installed system. For security reasons, modern systems rarely make this available. This seemingly trivial detail is hard to find out. Luckily, techniques have been created that make it possible to determine a system version remotely.

The stone age

Before we learn about the latest techniques and tools to remotely discover system versions, we should take a look at the methods used in the past.

Every server has services running that wait for a connection on a specific port. Otherwise they wouldn't be called servers. An example of such a service is the sshd server. It is the server for the SSH service; as standard it listens in on port 22. The majority of services, after making a connection, greet us with a welcome banner. This banner often contains a lot of useful information, for example the system version on which it is working. We will now try to connect with sshd running on the xxyyzz.com server using telnet:

```
bash-2.05b$ telnet xxyyzz.com 22
Trying 180.180.180.180...
Connected to xxyyzz.com.
Escape character is '^]'.
SSH-2.0-OpenSSH_3.5p1 FreeBSD-20090524
```

Using one simple procedure we have obtained a lot of interesting information. We know that the sshd version is OpenSSH_3.5p1 and that it

works under the FreeBSD system control edited on 2009.05.24. Now we can start searching for an exploit for OpenSSH on the FreeBSD system. Unfortunately, it is far from certain that we will find anything. This service, despite being a relatively old version, is considered reasonably secure. We can, however, check which other services are functioning, by looking at the server's banner.

Not every service supplies the system version in the welcome banner, but as we already know the version, the information about the version of the service alone should be sufficient for us when searching or writing appropriate exploits.

Unfortunately, the situation presented above happens only rarely. New service versions have ceased informing about the version of the system on which they work, because the less information made available, the safer it is. We will now attempt to connect with port 22 of a Mandrake server, a popular distribution of the Linux system:

```
bash-2.05b$ telnet www.mandrake.org 22
Trying 212.85.147.164.22...
Connected to 212.85.147.164.
Escape character is '^]'.
SSH-1.99-OpenSSH_3.6.1p2
```

We again received information about the service version, but we still don't know anything about the operating system. It is probably Linux, due to the content of the page itself, but unfortunately we cannot be sure of this. It could also be Solaris, HP-UX, or another system from the Unix family.

Our previous example, testing xxyyzz.com, also did not provide 100 percent certainty. The services allow us to change the welcome banner. A capable administrator will modify resources so that after connecting they will show fictitious information to mislead a potential hacker. Luckily, there are methods that require a lot of work on the part of the administrator to counteract, and using which we will be able to determine the version of the operating system without error.

OS fingerprinting

Each device working on the Internet must possess a code responsible for handling the network. This code, which is most frequently located in the system kernel, is called the network stack. There are many operating systems and many more people who have created them. Each operating system possesses a unique network stack written by different people. Observing the system behavior in the Internet we are able to determine its type. Network data are sent in packets. Each packet possesses a header, for example TCP/IP. The biggest differences between packets sent by various systems are located in the header. By establishing the characteristic behaviors of different operating systems we are able to distinguish them from one another, even on the basis of a single packet. We call this process OS fingerprinting. Luckily we do not have to examine the packets from different systems individually. We will use the large databases and special tools that have been created for this.

Active fingerprinting

Active fingerprinting is one variety of remote recognition of the system version. Active means that to obtain information it is required to generate traffic in the network. Tools using this method send packets of various kinds toward the server. Then they wait for an answer (or none) and examine the received results closely. In the next step they compare what they have observed with the information saved in the "fingerprints" database. While this is certainly effective, it also has its drawbacks. As mentioned above, to perform scanning of this kind we have to send data to the server. This entails the risk that our attack will be detected, warning the administrator that something is not right even before the target is attacked. Luckily, however, few servers possess services enabling them to detect these kinds of actions.

One of the most popular and best tools using this method is nmap. Almost all popular distributions of the Linux system include nmap as standard. If ours does not, we can also download it from the page:

```
http://www.nmap.org
```

After installing nmap we will now put it to use. The reader can find detailed information about the nmap application in the chapter dedicated to network scanners. We will perform our example tests on our local machine to avoid irritating any administrators.

At the beginning we scan the ports:

```
bash-2.05b$ nmap localhost

Starting nmap 5.00 ( http://www.insecure.org/nmap/ ) at 2010-01-11 21:10 CET
Interesting ports on localhost (127.0.0.1):
(The 1659 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 0.184 seconds
bash-2.05b$
```

As we can see, a few services that could constitute a potential danger are running on our example server. We use the -O option for nmap to show us the system version the localhost server is running on. However, to do this we have to have administrator privileges; otherwise, nmap won't be able to create a special socket that can send packets of various kinds.

```
bash-2.05b# nmap -O localhost

Starting nmap 5.00 ( http://www.insecure.org/nmap/ ) at 2010-01-11 21:13 CET
Interesting ports on localhost (127.0.0.1):
(The 1659 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

Device type: general purpose
Running: Linux 2.4.X|2.5.X|2.6.X
OS details: Linux 2.5.25 - 2.6.3 or Gentoo 1.2 Linux 2.4.19 rc1-rc7), Linux 2.6.3 -
2.6.8
Uptime 0.305 days (since Mon Jan 11 13:53:38 2010)

Nmap run completed -- 1 IP address (1 host up) scanned in 2.272 seconds
bash-2.05b#
```

This shows that the server is running the Linux system 2.4.X|2.5.X|2.6.X. Differences between the kernel branches in the code of the network service are small, so it is difficult to determine the exact version. Nmap attempts, however, to guess which system it exactly is:

```
OS details: Linux 2.5.25 - 2.6.3 or Gentoo 1.2 Linux 2.4.19 rc1-rc7), Linux 2.6.3-2.6.8
```

The real version of the system kernel running on the localhost computer is 2.6.7, which is within the range determined by the program. Nmap provides another interesting detail – the time the server has been running:

```
Uptime 0.305 days (since Mon Jan 11 13:53:38 2010)
```

The date is determined basing on the "Timestamp" field of the TCP/IP header. We already know how to use nmap to remotely recognize the system version, but we don't know how it really works. It is worth it to deepen our knowledge of this topic.

How it works

Nmap includes a database of fingerprints saved as standard in the file /usr/share/nmap/nmap-os-fingerprints. After performing a few tests, which we will discuss later on in this chapter, it compares the results with those available in the database. At the very top of the file, in the comments, is the instruction regarding the tests. The tests are called T1 (from TEST1), T2, and so on up to T7, as well as PU, which is a test of a different character. Below is a short description of each of these:

a) T1 sends a packet with the SYN flag set together with the TCP options on the open server port. The SYN flag set in the packet tells about the intention to make a connection with the server.
b) T2 sends an empty packet without any options on the open port.
c) T3 sends a packet with set flags SYN|FIN|URG|PSH on the open port without any options.
d) T4 sends a packet with ACK flag set on an open port without any options.
e) T5 sends a packet with SYN flag set on a closed port without any options.

f) T6 sends a packet with ACK flag set on a closed port without any options.
g) T6 sends a FIN|PSH|URG packet on a closed port without any options.
h) PU sends a UDP packet on a closed port.

As we can see it performs as many as eight tests to determine the system version. However, in order for the test to be terminated successfully, the server has to possess at least one open and one closed port. Otherwise the test won't be complete.

We will now analyze an example signature from the file nmap-os-fingerprints:

```
Fingerprint Linux 2.4.27 with grsec
Class Linux | Linux | 2.4.X | general purpose
TSeq(Class=TR%gcd=<6%IPID=RD%TS=1000HZ)
T1(DF=Y%W=16A0%ACK=S++%Flags=AS%Ops=MNNTNW)
T2(Resp=Y%DF=Y%W=0%ACK=S%Flags=AR%Ops=)
T3(Resp=Y%DF=Y%W=16A0%ACK=S++%Flags=AS%Ops=MNNTNW)
T4(DF=Y%W=0%ACK=O%Flags=R%Ops=)
T5(DF=Y%W=0%ACK=S++%Flags=AR%Ops=)
T6(DF=Y%W=0%ACK=O%Flags=R%Ops=)
T7(DF=Y%W=0%ACK=S++%Flags=AR%Ops=)
PU(DF=N%TOS=C0%IPLEN=164%RIPTL=148%RID=E%RIPCK=E%UCK=E%ULEN=134%DAT=E)
```

At first glance the signature may seem hard to interpret, because it is formatted for the program to process. Therefore we will now analyze each line in sequence:

```
Fingerprint Linux 2.4.27 with grsec
```

This states that the signature concerns the Linux system 2.4.27 with an installed grsecurity path (this path was discussed in the chapter on security paths). The second line has a bigger significance when searching for useful fingerprints:

```
Class Linux | Linux | 2.4.X | general purpose
```

This states, in sequence, that the signature belongs to the Linux class, Linux system, version 2.4.X, and that it is a universal signature. The next one contains the TSeq field:

```
TSeq(Class=TR%gcd=<6%IPID=RD%TS=1000HZ)
```

This tells us about the value of the TOS field of the IP header. The field depends on the device type from which it has been sent (e.g., it is different for the network card, modem, etc.).

```
T1(DF=Y%W=16A0%ACK=S++%Flags=AS%Ops=MNNTNW)
```

Here we have the answer to our first test. DF=Y, which means that the "Don't fragment" bit belonging to the IP header is set, and the packets can be divided into smaller ones. W=16A0 states that the "Window" field of the TCP header has the value 0x16A0. ACK=S++ tells us that the confirmation that we will receive (a reply on the SYN packet) must be an initiating sequence increased by one. Flags=AS means that the returned packet contains the ACK flags set, and SYN is a standard reply after sending a SYN packet on an open port. Ops=MNWNNT are the options of the TCP header in the following sequence:

```
Maximum segment size | NOP | Window size | NOP | NOP | Timestamp
```

These options are used when setting TCP/IP transmission parameters. They are used rarely and not all systems implement them. However, these options allow us to determine the version of the remote system with greater accuracy.

```
T2(Resp=Y%DF=N%W=0%ACK=S%Flags=AR%Ops=)
```

The majority of the tags used in the response are similar to the previous ones. Resp=Y means that we have to obtain a response to this test (which is not required, as we sent an empty packet). DF=N means that this time this bit is not set. Flags=AR tells that the return packet contains the ACK flags set and RESET. Ops= means that no options can be set.

```
T3(Resp=Y%DF=Y%W=16A0%ACK=S++%Flags=AS%Ops=MNNTNW)
```

As we can see, the response to this test is very similar to the one from T1. Here, however, the response (Resp=Y) is required, different flags are set (ACK and SYN), and the options are located in a different sequence (Ops=MNNTNW).

```
T4(DF=Y%W=0%ACK=O%Flags=R%Ops=)
```

As we already said, T4 is a test sending a packet from the set ACK flag on an open port. The server should respond to it with the packet with the set DF bit (DF=Y), Window field with the value 0 (W=0), confirmation equal to 0 (ACK=0) with the set RST flag (Flags=R), and without any options (Ops=).

```
T5(DF=Y%W=0%ACK=S++%Flags=AR%Ops=)
T6(DF=Y%W=0%ACK=0%Flags=R%Ops=)
T7(DF=Y%W=0%ACK=S++%Flags=AR%Ops=)
```

Here we have three consecutive tests that are very similar to each other. Each of them consists of sending packets on the closed port, but with other flags set. The responses are similar to those described before.

```
PU(DF=N%TOS=C0%IPLEN=164%RIPTL=148%RID=E%RIPCK=E%UCK=E%ULEN=134%DAT=E)
```

The next line describes a test with the UDP packet, sent on a closed port. The server should respond to it with a packet of the type "port unreachable." TOS=0 means that the "type of service" field of the IP header should have the value zero. IPLEN=164 tells us that the "IP total length" field has the value equal to 164. RIPTL=148 is the length of the packet without the IP header. RID=E means that a RID value is awaited in the packet responding to our packet. Whereas RIPCK=E states that the checksum of the packet is correct (in some cases this is not the case, and therefore it would be RIPCK=F). UCK=E tells us that the value of the checksum of the UDP header also is correct. ULEN=134 is the length of the UDP header and DAT=E means that the data in the UDP packet have been returned correctly.

Another tool using the active fingerprinting is the xprobe2 program. It can be downloaded from its publisher's page:

```
http://ofirarkin.wordpress.com/xprobe/
```

The compilation process of the sources looks much like it did in the case of other programs:

```
./configure && make && make install
```

After installation the program is ready to work. We will now log into the administrator's account and will scan the local machine:

```
bash-2.05b# xprobe2 localhost
Xprobe2 v.0.2.1 Copyright (c) 2002-2004 fyodor@o0o.nu, ofir@sys-security.com,
meder@o0o.nu

[+] Target is localhost
[+] Loading modules.
[+] Following modules are loaded:
[x] [1] ping:icmp_ping  -  ICMP echo discovery module
[x] [2] ping:tcp_ping  -  TCP-based ping discovery module
[x] [3] ping:udp_ping  -  UDP-based ping discovery module
[x] [4] infogather:ttl_calc  -  TCP and UDP based TTL distance calculation
[x] [5] infogather:portscan  -  TCP and UDP PortScanner
[x] [6] fingerprint:icmp_echo  -  ICMP Echo request fingerprinting module
[x] [7] fingerprint:icmp_tstamp  -  ICMP Timestamp request fingerprinting module
[x] [8] fingerprint:icmp_amask  -  ICMP Address mask request fingerprinting module
[x] [9] fingerprint:icmp_info  -  ICMP Information request fingerprinting module
[x] [10] fingerprint:icmp_port_unreach  -  ICMP port unreachable fingerprinting module
[x] [11] fingerprint:tcp_hshake  -  TCP Handshake fingerprinting module
[+] 11 modules registered
[+] Initializing scan engine
[+] Running scan engine
[-] ping:tcp_ping module: no closed/open TCP ports known on 127.0.0.1. Module test
failed
[-] ping:udp_ping module: no closed/open UDP ports known on 127.0.0.1. Module test
failed
[+] No distance calculation. 127.0.0.1 appears to be dead or no ports known
[+] Host: 127.0.0.1 is up (Guess probability: 25%)
[+] Target: 127.0.0.1 is alive. Round-Trip Time: 0.00018 sec
[+] Selected safe Round-Trip Time value is: 0.00037 sec
[-] fingerprint:tcp_hshake Module execution aborted (no open TCP ports known)
[+] Primary guess:
[+] Host 127.0.0.1 Running OS: "Linux Kernel 2.6.4" (Guess probability: 70%)
[+] Other guesses:
[+] Host 127.0.0.1 Running OS: "Linux Kernel 2.0.34" (Guess probability: 70%)
[+] Host 127.0.0.1 Running OS: "Linux Kernel 2.6.6" (Guess probability: 70%)
[+] Host 127.0.0.1 Running OS: "Linux Kernel 2.4.19" (Guess probability: 70%)
[+] Host 127.0.0.1 Running OS: "Linux Kernel 2.6.8" (Guess probability: 70%)
[+] Host 127.0.0.1 Running OS: "Linux Kernel 2.4.21" (Guess probability: 70%)
[+] Host 127.0.0.1 Running OS: "Linux Kernel 2.4.21" (Guess probability: 70%)
[+] Host 127.0.0.1 Running OS: "Linux Kernel 2.6.8" (Guess probability: 70%)
[+] Host 127.0.0.1 Running OS: "Linux Kernel 2.4.19" (Guess probability: 70%)
[+] Host 127.0.0.1 Running OS: "Linux Kernel 2.6.6" (Guess probability: 70%)
[+] Cleaning up scan engine
[+] Modules deinitialized
[+] Execution completed.
bash-2.05b#
```

At the beginning we see information about the uploaded modules used when scanning. Much of it regards the ICMP scanning. It is the main characteristic of this program. As opposed to nmap, which uses almost exclusively TCP/IP,

xprobe2 focuses on tests related to the ICMP protocol. The authors of this program also created the project "ICMP usage in scanning." We can read about this on the page:

```
http://ofirarkin.files.wordpress.com/2008/11/icmp_scanning_v30.pdf
```

After performing all tests, xprobe2 gives us the system version. It is probably Linux kernel 2.6.4, but it indicates also other possibilities. As we can see, there are many of them and the range for the possible version of the Linux system is very wide (from 2.0.34 to 2.6.8). The probability of this being the version is also underwhelming (70%). Analyzing what the program returned, we see a few details about the failure:

```
[-] ping:tcp_ping module: no closed/open TCP ports known on 127.0.0.1. Module test
failed
[-] ping:udp_ping module: no closed/open UDP ports known on 127.0.0.1. Module test
failed
...
[-] fingerprint:tcp_hshake Module execution aborted (no open TCP ports known)
```

As we can see, three modules terminated with an error. The reason was that they didn't know the open ports of the server. A similar situation is to be found in nmap – it requires one open and one closed port to examine the system version correctly. Nmap deals with this by scanning the ports of the host under attack. Xprobe2 also possesses such a function, however, it is better to use the -p option, which allows the ports to be determined independently. It assumes the parameters in the following form:

```
protocol:port:open/closed
```

We want to tell the program that the port 22 TCP is open, 21 TCP is closed, and 53 UDP is also closed.

Let's see how the program will react in this case:

```
bash-2.05b# xprobe2 -p tcp:22:open -p tcp:21:closed -p udp:53:closed localhost

Xprobe2 v.0.2.1 Copyright (c) 2002-2004 fyodor@o0o.nu, ofir@sys-security.com,
meder@o0o.nu

[+] Target is localhost
```

```
[+] Loading modules.
[+] Following modules are loaded:
[x] [1] ping:icmp_ping  -  ICMP echo discovery module
[x] [2] ping:tcp_ping  -  TCP-based ping discovery module
[x] [3] ping:udp_ping  -  UDP-based ping discovery module
[x] [4] infogather:ttl_calc  -  TCP and UDP based TTL distance calculation
[x] [5] infogather:portscan  -  TCP and UDP PortScanner
[x] [6] fingerprint:icmp_echo  -  ICMP Echo request fingerprinting module
[x] [7] fingerprint:icmp_tstamp  -  ICMP Timestamp request fingerprinting module
[x] [8] fingerprint:icmp_amask  -  ICMP Address mask request fingerprinting module
[x] [9] fingerprint:icmp_info  -  ICMP Information request fingerprinting module
[x] [10] fingerprint:icmp_port_unreach  -  ICMP port unreachable fingerprinting module
[x] [11] fingerprint:tcp_hshake  -  TCP Handshake fingerprinting module
[+] 11 modules registered
[+] Initializing scan engine
[+] Running scan engine
[+] Host: 127.0.0.1 is up (Guess probability: 75%)
[+] Target: 127.0.0.1 is alive. Round-Trip Time: 0.01541 sec
[+] Selected safe Round-Trip Time value is: 0.03082 sec
[+] Primary guess:
[+] Host 127.0.0.1 Running OS: "Linux Kernel 2.6.1" (Guess probability: 94%)
[+] Other guesses:
[+] Host 127.0.0.1 Running OS: "Linux Kernel 2.4.22" (Guess probability: 94%)
[+] Host 127.0.0.1 Running OS: "Linux Kernel 2.4.28" (Guess probability: 94%)
[+] Host 127.0.0.1 Running OS: "Linux Kernel 2.4.24" (Guess probability: 94%)
[+] Host 127.0.0.1 Running OS: "Linux Kernel 2.4.26" (Guess probability: 94%)
[+] Host 127.0.0.1 Running OS: "Linux Kernel 2.4.26" (Guess probability: 94%)
[+] Host 127.0.0.1 Running OS: "Linux Kernel 2.4.24" (Guess probability: 94%)
[+] Host 127.0.0.1 Running OS: "Linux Kernel 2.4.28" (Guess probability: 94%)
[+] Host 127.0.0.1 Running OS: "Linux Kernel 2.4.22" (Guess probability: 94%)
[+] Host 127.0.0.1 Running OS: "Linux Kernel 2.6.1" (Guess probability: 94%)
[+] Cleaning up scan engine
[+] Modules deinitialized
[+] Execution completed.
bash-2.05b#
```

This time we didn't obtain any information about errors. The scope of the searched system version also decreased (2.4.22 - 2.6.1) which means that scanning was more accurate. The probability of it being exactly this system version is now quite good (94%). Nevertheless, nmap was closer to the truth anyway. The TCP/IP tests allow more accurate examination of the network stack, but are easier to be detected.

Xprobe2 also possesses also an option enabling us to scan ports. It is sufficient to use the -T option, and to enter the scope of the ports to be scanned in the parameter. We are usually required to have only one open port and one closed port. We know that on localhost sshd is running on port 22, and port 21 is closed. Therefore, we give this information to the program to save time in scanning the remaining ports. The result will be exactly the

same as with entering ports statically in the command line. The program will, however, generate quite a lot of unnecessary packets, which entails a bigger probability of our experiment being detected.

We now know exactly how active fingerprinting works. As we can see it is a complex process and very detailed. Another more commonly used type of remote recognition of the system version is passive fingerprinting.

Passive fingerprinting

Passive fingerprinting is characterized by a lack of generated network traffic. Like active fingerprinting, it involves observing packets, but it does not create traffic. This is a big advantage if our target requires greater caution than usual. The tools in use for passive fingerprinting work as servers. They intercept packets as popular sniffers do, but instead of passwords, they search within the packets for information on the system version. The tools use the same differences in the implementation of the network stack. Their signatures differ slightly, however, from those known already from nmap.

The most popular tool for passive fingerprinting is the p0f application. This project is being developed and the signature database is constantly increasing. It even contains the signatures of such devices using the Internet as the PlayStation2 console and Nokia telephones. This program isn't yet a part of popular distributions, however, we hope this will change in the future. Therefore we have to reach for the newest sources, which we will download from the project page:

```
http://lcamtuf.coredump.cx/p0f.shtml
```

The newest stable version can always be found under the address:

```
http://lcamtuf.coredump.cx/p0f.tgz
```

The download and compilation process is visible below:

```
bash-2.05b$ wget http://lcamtuf.coredump.cx/p0f.tgz
bash-2.05b$ tar zxvf p0f.tgz 1>/dev/null
```

```
bash-2.05b$ cd p0f
bash-2.05b$ make
bash-2.05b# make install
```

P0f uses the libpcap library, so if we don't have it, we have to install it. After compilation the program is ready to work. It creates a listening-in raw socket, for which administrator's privileges are required. We will now attempt to start up the program:

```
bash-2.05b# p0f
p0f - passive os fingerprinting utility, version 2.0.5
(C) M. Zalewski <lcamtuf@dione.cc>, W. Stearns <wstearns@pobox.com>
p0f: listening (SYN) on 'eth0', 231 sigs (13 generic), rule: 'all'.
```

Nothing more is happening. The program is currently waiting for packets with which it can determine the system version. Let's assume that we want to check, using p0f, which system is located on the server xxyyzz.com. Unfortunately, the server won't send us any packets by itself. Therefore, we have to do something; for example, to enter the server's WWW page. It will definitely be less suspicious the scanning with nmap. We will now wander around on the page and then return to the console. We now see what happened:

```
192.168.0.158:2921 - Linux 2.6.8 and newer (?) (up: 88 hrs)
  -> 180.180.180.180:80 (distance 0, link: ethernet/modem)
192.168.0.158:2922 - Linux 2.6.8 and newer (?) (up: 88 hrs)
  -> 180.180.180.180:80 (distance 0, link: ethernet/modem)
192.168.0.158:2923 - Linux 2.6.8 and newer (?) (up: 88 hrs)
  -> 180.180.180.180:80 (distance 0, link: ethernet/modem)
192.168.0.158:2924 - Linux 2.6.8 and newer (?) (up: 88 hrs)
  -> 180.180.180.180:80 (distance 0, link: ethernet/modem)
192.168.0.158:2925 - Linux 2.6.8 and newer (?) (up: 88 hrs)
  -> 180.180.180.180:80 (distance 0, link: ethernet/modem)
```

We see that some connections have been made from 192.168.0.158 (that's us) to 180.180.180.180 (IP number of the host xxyyzz.com). P0f showed also that the system version running on our computer is Linux 2.6.8. Although this is not quite correct, it is a much better result than that given by nmap. However, we don't know which system is running on xxyyzz.com, even though obtaining this information is our main goal. P0f didn't recognize it, because

we started it up in the default mode of listening for the SYN packets. These packets are sent only to attempt to make a connection. We connect to the server and send them, and not the other way round. After sending the packet with the SYN flag set, the server responds with the packet SYN+ACK (agreeing to the connection). Luckily, p0f also includes signatures for this type of packet. We simply start it with the -A option. Let's do this and open the target WWW page again:

```
bash-2.05b# p0f -A
p0f - passive os fingerprinting utility, version 2.0.5
(C) M. Zalewski <lcamtuf@dione.cc>, W. Stearns <wstearns@pobox.com>
p0f: listening (SYN+ACK) on 'eth0', 57 sigs (1 generic), rule: 'all'.
180.180.180.180:80 - FreeBSD 4.6-4.8 (RFC1323) (up: 226 hrs)
  -> 192.168.0.158:2914 (distance 16, link: ethernet/modem)
180.180.180.180:80 - FreeBSD 4.6-4.8 (RFC1323) (up: 226 hrs)
  -> 192.168.0.158:2915 (distance 16, link: ethernet/modem)
180.180.180.180:80 - FreeBSD 4.6-4.8 (RFC1323) (up: 226 hrs)
  -> 192.168.0.158:2916 (distance 16, link: ethernet/modem)
180.180.180.180:80 - FreeBSD 4.6-4.8 (RFC1323) (up: 226 hrs)
  -> 192.168.0.158:2917 (distance 16, link: ethernet/modem)
```

This time we have the reverse situation. We don't know our system version, but we know where the target server is running. It is FreeBSD of a version from 4.6 to 4.8. Our previous tests detected similar information and the nmap program would tell us much the same.

In the current version, p0f is not able to take into consideration many kinds of packets simultaneously. This is, however, one of the purposes of its development, so we can expect that this option will be added in future versions. P0f is an excellent tool for administrators in consumer networks. When started up on a server it can, for example, detect which users are sharing a link, in violation of their contract. If packets characteristic of many systems (e.g. Linux and Windows) come from one IP address, P0f will notify the administrator about this, showing beyond doubt that a link is being shared. This provides effective interception and cut-off of dishonest clients.

We have managed to exploit p0f to determine the system version of the remote server without generating any suspicious traffic. We will now check if the way in which it examines the packets differs much from that of nmap.

The p0f configuration files are located as standard in the /etc/p0f folder. These are:

a) p0f.fp – This file contains the signatures of the fingerprints for the TCP/IP packets with the SYN flag set.
b) p0fa.fp – This contains signatures for the packets SYN+ACK. As we know, p0f uses this database when it is started with the -A option.
c) p0fo.fp – This is a fingerprint database for packets with the ACK flag set (that is, for the confirming packets).
d) p0fr.fp - List of signatures for packets with the RST or RST+ACK flag set. Such packets are sent when our packet will be, for example, rejected by the firewall or when the connection won't be able to be made.

Let's take a look at an example SYN signature:

```
65535:64:1:60:M*,N,W2,N,N,T:Z:FreeBSD:5.1-current (3)
```

The first value (65535) is the window size; that is, the Window field of the TCP header. Then we have the value of the TTL field (time to live), or the lifetime of the packet (64 jumps). The third value means the bit doesn't fragment, which we mentioned while discussing nmap (1 = set). The next number is the total size of the SYN packet (headers + options of the TCP packet). Then the options of the TCP header follow:

```
M*,N,W2,N,N,T
```

These are, in sequence:

a) Maximum size of the segment with any value (M*).
b) NOP, or an empty field (N).
c) Window scale option with the value 2 (W2).
d) NOP (N).
e) NOP (N).
f) Timestamp, usually with the value of system functioning time, thanks to which we can remotely find out its uptime.

The following field equal to the letter "Z" tells us that the value of the Id field of the IP header is equal zero. In this place also other letters can appear, and they notify about strange behavior of the network stack. Their full description is located in the p0f.fp file in the comprehensive commentary. Next, we have already the version of the specific system, that is FreeBSD:5.1-current (3). As we can see, p0f mostly uses the same methods as nmap to determine the true system version. The signatures for the packets SYN+ACK, ACK, and RST+ present themselves in exactly the same way. The only differences are other anomalies related to the network stack; they are, however, described in detail in each database.

Exploiting p0fa against nmap

Nmap is an active scanner, and as we already have said, it generates quite a lot of packets to achieve its purpose. P0f only listens in and observes. However, it turns out that it can be used by us also as defense against nmap. There are some interesting positions in the SYN p0fa signature database:

```
1024:64:0:40:.:.:-*NMAP:syn scan (1)
2048:64:0:40:.:.:-*NMAP:syn scan (2)
3072:64:0:40:.:.:-*NMAP:syn scan (3)
4096:64:0:40:.:.:-*NMAP:syn scan (4)

1024:64:0:40:.:A:-*NMAP:TCP sweep probe (1)
2048:64:0:40:.:A:-*NMAP:TCP sweep probe (2)
3072:64:0:40:.:A:-*NMAP:TCP sweep probe (3)
4096:64:0:40:.:A:-*NMAP:TCP sweep probe (4)

1024:64:0:60:W10,N,M265,T,E:P:-*NMAP:OS detection probe (1)
2048:64:0:60:W10,N,M265,T,E:P:-*NMAP:OS detection probe (2)
3072:64:0:60:W10,N,M265,T,E:P:-*NMAP:OS detection probe (3)
4096:64:0:60:W10,N,M265,T,E:P:-*NMAP:OS detection probe (4)

1024:64:0:60:W10,N,M265,T,E:PF:-*NMAP:OS detection probe w/flags (1)
2048:64:0:60:W10,N,M265,T,E:PF:-*NMAP:OS detection probe w/flags (2)
3072:64:0:60:W10,N,M265,T,E:PF:-*NMAP:OS detection probe w/flags (3)
4096:64:0:60:W10,N,M265,T,E:PF:-*NMAP:OS detection probe w/flags (4)
```

These are all characteristic for nmap packets. Thanks to these, p0f can detect scanning and an attempt of system detecting. We shall check to see if it indeed works.

We start up p0fa on one console:

```
bash-2.05b# p0f -i lo
p0f - passive os fingerprinting utility, version 2.0.5
(C) M. Zalewski <lcamtuf@dione.cc>, W. Stearns <wstearns@pobox.com>
p0f: listening (SYN) on 'lo', 231 sigs (13 generic), rule: 'all'.
```

The -i option tells the program which interface it should listen in on. In our case it is the local interface (lo). Now we go to the second console and run:

```
bash-2.05b# nmap localhost

Starting nmap 5.00 ( http://www.insecure.org/nmap/ ) at 2010-01-12 14:11 CET
Interesting ports on localhost (127.0.0.1):
(The 1659 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 0.547 seconds
bash-2.05b#
```

We have scanned the ports of the local machine. Let's check the reaction of p0fa:

```
...
127.0.0.1:40058 - NMAP syn scan (2) *
127.0.0.1:40058 - NMAP syn scan (1) *
127.0.0.1:40058 - NMAP syn scan (3) *
127.0.0.1:40058 - NMAP syn scan (4) *
127.0.0.1:40058 - NMAP syn scan (1) *
127.0.0.1:40058 - NMAP syn scan (2) *
127.0.0.1:40058 - NMAP syn scan (1) *
127.0.0.1:40058 - NMAP syn scan (4) *
127.0.0.1:40058 - NMAP syn scan (2) *
127.0.0.1:40058 - NMAP syn scan (1) *
127.0.0.1:40058 - NMAP syn scan (2) *
127.0.0.1:40058 - NMAP syn scan (4) *
...
```

We have seen a large number of examined syn scan packets (they are used by nmap to scan ports). We know the exact IP of the attacker (127.0.0.1), which can be helpful in the later potential analysis of the cracking.

Let's not switch off p0fa and instead order nmap to check our system version:

```
bash-2.05b# nmap -O localhost

Starting nmap 5.00 ( http://www.insecure.org/nmap/ ) at 2010-01-12 14:14 CET
Interesting ports on localhost (127.0.0.1):
(The 1659 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

Device type: general purpose
Running: Linux 2.4.X|2.5.X|2.6.X
OS details: Linux 2.5.25 - 2.6.3 or Gentoo 1.2 Linux 2.4.19 rc1-rc7), Linux 2.6.3 -
2.6.8
Uptime 0.044 days (since Tue Jan 12 13:11:27 2010)

Nmap run completed -- 1 IP address (1 host up) scanned in 2.560 seconds
bash-2.05b#
```

Before checking the system version, nmap must scan our ports in a way to find at least one open and one closed. We will now check the reaction of p0f:

```
...
127.0.0.1:35157 - NMAP syn scan (1) *
127.0.0.1:35157 - NMAP syn scan (4) *
127.0.0.1:35157 - NMAP syn scan (3) *
127.0.0.1:35157 - NMAP syn scan (3) *
127.0.0.1:35164 - NMAP OS detection probe (3) (ECN) *
127.0.0.1:35168 - NMAP OS detection probe (3) *
127.0.0.1:35158 - NMAP OS detection probe (1) *
127.0.0.1:35159 - NMAP OS detection probe (4) *
127.0.0.1:35160 - NMAP OS detection probe (2) *
127.0.0.1:35161 - NMAP OS detection probe (3) *
127.0.0.1:35162 - NMAP OS detection probe (3) *
127.0.0.1:35163 - NMAP OS detection probe (4) *
```

At the beginning we see a lot of syn scan packets. Next, p0f examines eight packets that help nmap to remote detect the system version. There are exactly as many as the tests performed. After this type of scanning we can block the attacker with the firewall so that he will give up his penetration tests. As we can see, the tool serving to attack can also be used as a detecting system to detect cracking or illegal link sharing.

We managed to detect the system version of the remote server using various methods for this purpose. As we can see, detecting fingerprints is not

difficult; we just need to use another tool taking advantage of the same method. Now we know that nmap scanners cannot hide from detection, just as operating systems cannot. Each possesses a characteristic way of functioning. An administrator can try to change individual elements of outgoing packets, but this can cause the network to function poorly, and does not give any guarantee of security. Therefore, despite hiding details regarding the server, it is better to take appropriate security measures so that, even after receiving such information, the server will continue to be safe. The reader can read more about packet behavior in the chapter on Netfilter.