Chapter 15

Netfilter and system security services

The kernel of the Linux system possesses a mechanism that allows us to add new abilities even after compiling. This is possible because of its modular structure. We can upload the kernel module to extend the abilities of our system at any time. With control over the kernel network layer we can modify all the packets that pass through our machine. This opens up huge possibilities that the reader will discover in this chapter.

Fingerprinting: a recap

When attempting to crack a computer, the first thing that a hacker does it to make a list of all active services on the server under attack; most frequently he uses a ready-made tool such as nmap for this purpose. If it turns out that one of the services is full of holes, it creates an opportunity to create an exploit that will work with the shellcode and its address in memory. This is different for each specific version of the operating system. The attacker therefore has only one chance to send and start up a shellcode compatible with the system. If this fails, the attack will probably cause the service to shut down. The matter of the shellcode address in memory seems to be quite similar. And likewise, without knowing the type of system working on the server, the hacker can only dream about executing a successful attack. It sometimes happens that the headers of the remote daemons reveal the version of the system on which they are running. The example below shows how the exact version of the OpenSSH server and the type of the system on which it is running can be obtained without great difficulty using telnet:

```
[dave@polygon ~]>> telnet xxyyzz.org 22
Trying 62.62.62.62...
```

```
Connected to www.xxyyzz.com.com.
Escape character is '^]'.
SSH-1.99-OpenSSH_3.5p1 FreeBSD-20090524
```

Identification of this type is unfortunately less and less common. Software producers often refrain from providing specific information in headers, or they make the headers easy to modify, as we have already mentioned in the previous chapter. Remote identification of the system can be difficult. Here, what is known as "stack fingerprinting" can be helpful. This is a method allowing us to remotely detect the system version by examining the behavior of the network stack of the scanned machine. Apart from a description of the required part of the network stack, the standards contained in all machines working on the Internet also mention things that are not necessary for implementation. An example might be certain options of the TCP protocol. This causes certain systems to behave slightly differently in the network. It is exactly this fact that makes the program suitable for remote fingerprinting use. Having at his disposal one packet sent from the attacked machine, the hacker is able to determine the system version, or even the time it was running. The most popular tools that take advantage of these methods are nmap and p0f. These tools can be downloaded from the pages:

```
http://www.insecure.org/nmap
http://lcamtuf.coredump.cx/p0f.shtml
```

As we have already said, they represent two basic kinds of remote fingerprinting – active and passive. Nmap is an active scanner, meaning that it sends specific packets to examine the network stack of the remote computer. Let's see what it will tell us about the server xxyyzz.org.

```
[dave@polygon ~]>> nmap -O -p 20-22 xxyyzz.org

Starting nmap 5.00 ( http://www.insecure.org/nmap/ ) at 2010-05-28 21:37 CEST
Interesting ports on www.xxyyzz.com (62.62.62.62):
PORT             STATE            SERVICE
20/tcp           closed           ftp-data
21/tcp           open             ftp
22/tcp           open             ssh

Device type: general purpose
Running: FreeBSD 4.X
```

```
OS details: FreeBSD 4.6.2-RELEASE - 4.8-RELEASE, FreeBSD 4.7-RELEASE, FreeBSD 4.7-
RELEASE-p3, FreeBSD 4.8-STABLE
Uptime 21.123 days (since Mon Jun 7 18:40:29 2010)

Nmap run completed -- 1 IP address (1 host up) scanned in 11.036 seconds
```

The result is, of course, not a surprise. This method is easy to detect, and therefore, passive fingerprinting is used more frequently. It works by intercepting packets that are circulating in the network to determine the operating system version. p0f is one example of a tool that does this. It doesn't generate any traffic and therefore it is impossible to detect. However, there are methods that can secure us against this technique. One of them is to modify the network stack in such a way that it resembles one implemented in another system. This is possible thanks to a kernel component called Netfilter, which enables us to manipulate network behavior.

Our main task will be to overwrite the module, thus misleading the tools that are used to remotely identify the system version. At the beginning, however, it is worth taking a closer look at the structure of the kernel modules and at the function of Netfilter itself. More about fingerprinting can be found in the chapter dealing with remote identification of the operating system version.

Kernel modules

This chapter is not a course on writing modules but a description of their practical application in the kernel network layer. We assume that the reader already has experience in programming using the system kernel. We will, however, briefly describe the way they are created. However, there is more information about the kernel modules in the chapter dealing with hiding processes.

The kernel modules differ slightly from common programs written in the C language. They don't use the standard libc headers, but ones located in the kernel resources. In keeping with tradition, our first module will print the words "Hello world!" on the screen (**/CD/Chapter15/Listings/modul.c**).

```
/* Listing 0. example kernel module */
#include <linux/module.h>
#include <linux/init.h>
```

```
#include <linux/kernel.h>

MODULE_LICENSE ("GPL");

/* module initiation */
int  __init mod_init()
{
         printk("<1>Hello world!");
         return 0;
}

/* module unloading */
void __exit mod_exit()
{

}

module_init(mod_init);
module_exit(mod_exit);
```

We can describe the mod_init() function as an equivalent of the main() function. After uploading the module, its execution begins exactly from this function. The only thing the above module does it to put the string "Hello world!" into the kernel log. The mod_exit function is executed while unloading the module. To compile the examples shown in this chapter we will use the Makefile file.

```
obj-m := modul.o modul2.o modul3.o modul4.o modul5.o modul6.o
all:
make -C /usr/src/linux SUBDIRS=${PWD}
clean:
rm -f *.o *.tmp .*.o.* .tmp_versions/* .*.ko.cmd *.mod.c Module.symvers \
modules.order
rmdir .tmp_versions
```

The above script compiles a file with the name "modul.c." In order for the compilation to succeed for a 2.6 kernel, we have to have the system resources located in the folder /usr/src/linux. The "insmod" program loads the module and "rmmod" unloads it. Depending on the system version, the output file will possess the extension .o (2.4 kernels) or .ko (2.6 kernels). We will now try to test our first kernel module.

```
bash-2.05b# make -s
bash-2.05b# insmod modul.o
bash-2.05b# dmesg | tail –n 5
...
Hello world!
bash-2.05b# rmmod modul
```

After uploading and performing the mod_init function, our module doesn't do anything. In the following part of the chapter we will focus on modules exploiting the part of the kernel responsible for network services, more specifically Netfilter.

Netfilter

Netfilter is a subsystem in the system kernel that enables us to check packet headers and to decide about the fate of the packets while they are stored on the machine on which Netfilter is running. This subsystem has at its disposal several simple macros regarding the packets that pass through. The packet can be accepted NF_ACCEPT), rejected (NF_DROP), or "stolen" (NF_STOLEN). These macros are defined in the file (linux/netfilter.h).

The manipulation of the packets takes place using five predefined macros:

- **NF_IP_PRE_ROUTING** - before we make a decision about the packet's fate; whether it should pass through or be rejected.
- **NF_IP_LOCAL_IN** - we allow packets intended for our host to pass through.
- **NF_IP_FORWARD** - we allow the packet to pass through if it is intended for another host or interface.
- **NF_IP_LOCAL_OUT** - we filter the outgoing packets if they come from a local process.
- **NF_POST_ROUTING** - we filter the outgoing packets.

All packets passing through the device are held in a structure called sk_buff. This structure is defined in the file linux/skbuff.h; it is the buffer in which the system kernel stores packets. When the network card receives a packet, it sends it to sk_buff and it transfers it to the network stack, using sk_buff the whole time.

```
        - next            -> pointer to the buffer on the list;
        - prev            -> pointer to the previous buffer on the list;
        - list            -> list, on which we are;
        - sk              -> socket, to which we belong;
        - stamp           -> time, in which we arrived at the host;
        - dev             -> device, with which we left the host;
        - rx_dev -> device, with which we arrived;
```

```
        - h             -> header of the transport layer (tcp, udp, icmp, igmp, raw);
        - nh            -> header of the network layer (ipv4, ipv6, ipx, raw);
        - mac           -> header of the physical layer;
        - cb            -> control buffer, used internally;
        - len           -> total length of the packet data;
        - csum          -> packet checksum;
        - pkg_type      -> packet type;
        - head          -> pointer to the header;
        - data          -> pointer to the data;
```

There are still other fields but from our point of view they are superfluous, so we won't describe them.

To use Netfilter in your own module it is necessary to create a hook function for it. Such a function first has to be registered. To register hooks, a structure called nf_hook_ops is used. This is defined in the file /usr/include/linux/netfilter.h and has the following form:

```
struct nf_hook_ops
{
        struct list_head list;
        nf_hookfn *hook;
        int pf;
        int hooknum;
        int priority;
};
```

A hook is a pointer to the function that will be responsible for the filtration of the packets. The pf field determines the protocol family. The protocol families available are defined in the file /usr/include/linux/socket.h. However, for IPv4 we should use PF_INET. The hooknum field specifies the level on which the decisions about the packet in the function we create should take place. In the end, the field priority determines where a given hook should be put. For the IPv4 protocol, the available values are defined in the file /usr/include/linux/netfilter_ipv4.h, in the field nf_ip_hook_priorities. In the example module we will use the macro NF_IP_PRI_FIRST.

The hook registration requires using the function nf_register_hook(), which assumes as a parameter the address of the structure nf_hook_ops and returns the int value. However, when we take a closer look at the function nf_reginster_hook(), located in the file net/core/netfilter.c, we will notice that it always returnes the value 0, whereas nf_unregister_hook() releases the

function and as parameter it also collects the address for the structure nf_hook_ops. We have to call it when unloading the module. In creating the filtering function, we therefore have to declare the structure nf_hook_ops, fill it in, and call the function nf_register_hook(). Below is an example of registration of the function with the name "hook" that filters all incoming packets.

```
static struct nf_hook_ops nfho;

nfho.hook                = hook;
nfho.hooknum             = NF_IP_PRE_ROUTING;
nfho.pf                  = PF_INET;
nfho.priority            = NF_IP_PRI_FIRST;

nf_register_hook(&nfho);
```

All declared functions hooking Netfilter have the following form:

```
unsigned int function_name(unsigned int hooknum,
                           struct sk_buff *sb,
                           const struct net_device *in,
                           const struct net_device *out,
                           int (*okfn)(struct sk_buff *))
```

The first argument of the function is the macro responsible for the level on which the decision about the packet's fate is made (as described above). The second argument is the pointer to the sk_buff structure. Two consecutive parameters are the pointers to the net_device structure. The net_device structure is used by the kernel for the description of all kinds of network interfaces. The first of those pointers refers to the "in" field, which describes the interface on which the packets are received, while the "out" field describes the interface from which the packets are sent. It is important to realize that often only one field will be available. For example the "in" field will be available only for hooks using the macros NF_IP_PRE_ROUTING and NF_IP_LOCAL_IN, while "out" will be available for those with the macros NF_IP_LOCAL_OUT and NF_IP_POST_ROUTING.

Attention.
In the recent 2.6 Linux kernels the definition of NF_IP_PRE_ROUTING and similar macros are not available for applications running in kernel mode. To be able to benefit from these definitions, we need to define them in the code

by ourselves or comment out the conditions ifndef __KERNEL__ / endif in the /usr/include/linux/netfilter_ipv4.h file. We will add the definition to our program manually (#define NF_IP_PRE_ROUTING 0).

On the listing below there is a simple Netfilter module that rejects all incoming packets. This example shows also how the values returned by the function (macros deciding about the packet fate) are interpreted by the module.

Here is the source code of the module rejecting all incoming packets (**/CD/Chapter15/Listings/modul2.c**):

```
/* Example code rejecting all incoming packets */

#include <linux/module.h>
#include <linux/kernel.h>
/* Netfilter header files */
#include <linux/netfilter.h>
#include <linux/netfilter_ipv4.h>

MODULE_LICENSE ("GPL");

/* declaration of the structure we use for registration of our function */
static struct nf_hook_ops nfho;

/* declaration of the function being hooked */
unsigned int hook(unsigned int hooknum,
                          struct sk_buff **sb,
                          const struct net_device *in,
                          const struct net_device *out,
                          int (*okfn)(struct sk_buff *))
{
        return NF_DROP;           /* reject all packets */
}

/* module initiation */
int __init mod_init()
{
        /* filling in the structure */
        nfho.hook = hook;         /* handle for our function */

/* Used before making decision about the packet (to pass through or not) */
        nfho.hooknum     = NF_IP_PRE_ROUTING;
        nfho.pf          = PF_INET; /* address family */
        /* setting the priority of our function to one */
        nfho.priority    = NF_IP_PRI_FIRST;
        nf_register_hook(&nfho); /* function registration */
```

```
        return 0;
}

/* module unloading */
void __exit mod_exit()
{
        nf_unregister_hook(&nfho); /* function release */
}

module_init(mod_init);
module_exit(mod_exit);
/* listing end */
```

After uploading this module all connections are cut and all incoming packets are automatically rejected.

Filtration of packets

Netfilter enables the filtering of packets that meet specific criteria. Using the name field from the net_device structure we can filter packets based on the interface from which they come. To reject the packets incoming on "eth0" the only activity performed is checking the value of the in->name field. If the value of the field is equal to "eth0" the function should return the macro NF_DROP. The source of the next listing presents a simple example of the use of this technique, showing filtering based on interface (**/CD/Chapter15/Listings/modul3.c**):

```
1| /* Interface, that we filter */
2| #define IFACE " eth0 "
3|
4| unsigned int hook(unsigned int hooknum,
5| struct sk_buff *sb,
6| const struct net_device *in,
7| const struct net_device *out,
8| int (*okfn)(struct sk_buff *))
9| {
10|
11|      // we don't want any null pointers
12|       if (in->name == NULL)
13|               return NF_ACCEPT;
14|
15|       if (in->name == „ ")
16|                return NF_ACCEPT;
17|
18|      // We check if packet passed through our interface
```

```
19|      if (strcmp(in->name, IFACE) == 0)
20|      {
21|              printk ("Packet rejected on interface:  %s.\n", IFACE);
22|              return NF_DROP;
23|      }
24|      else
25|              return NF_ACCEPT;
26|}
```

After uploading the module its function can be checked in the following way:

```
bash-2.05b# insmod modul3.ko
bash-2.05b# ping IP_ADDRESS_ON_ETH0
bash-2.05b# dmesg | tail -n 1
[*] Packet rejected on interface: eth0
bash-2.05b#
```

You must give your own IP address so the packets will be transferred to the interface connected with the address. After the interface is changed to "lo" it can be referred to the localhost.

In the system logs we will see the entry:

```
[*] Packet rejected on interface: eth0
```

Port filtering

Another simple filtration method is through the TCP target port of the packet. This method is slightly more complicated than the previous one, because we have to create the pointer to the TCP header independently. This is a simple operation consisting of creating a pointer to the tcphdr structure (defined in the file /usr/include/linux/tcp.h) right after the pointer to the IP header in our packet. We do this as follows:

```
iph = (struct iphdr *)skb->network_header;
tcph = (struct tcphdr *)(skb->data + (iph->ihl * 4));
```

Our filtering module will also take into consideration the source IP addressees of the packets, using which we will create a simple firewall.

Here is the source code of a ready-made filtering module based on the port and the IP address (**/CD/Chapter15/Listings/modul4.c**):

```
1| #include <linux/module.h>
2| #include <linux/init.h>
3| #include <linux/kernel.h>
4| #include <linux/skbuff.h>
5| #include <linux/tcp.h>
6| #include <linux/ip.h>
7| #include <linux/in.h>
8| #include <linux/netfi lter.h>
9| #include <linux/netfi lter_ipv4.h>
10|
11| #define NF_IP_PRE_ROUTING 0
12| // IP address you want to reject connections for
13| #define DROP_IP „192.168.56.1”
14|
15| MODULE_LICENSE(„GPL”);
16|
17| static struct nf_hook_ops nfho;
18|
19| // Convert IP address in order to be intelligible by the system
20| // (network byte order) - author of the code: Paolo Ardoino
21| static unsigned long my_addr(char *ip)
22| {
23|   unsigned long tmp;
24|   unsigned int val;
25|   int ctr;
26|
27|   tmp = 0;
28|   for (ctr = 0; ctr < 4; ctr++) {
29|     tmp <<= 8;
30|     if(*ip != ‘\0’) {
31|       val = 0;
32|       while (*ip != ‘\0’ && *ip != ‘.’) {
33|         val *= 10;
34|        val += *ip - ‘0’;
35|        ip++;
36|      }
37|      tmp |= val;
38|      if (*ip != ‘\0’) {
39|        ip++;
40| }
41| }
42| }
43| return htonl(tmp);
44| }
45|
46| // Convert IP address to string form
47| // - author of the code: Paolo Ardoino
48| static char *my_ntoa(unsigned long addr)
49| {
50| static char buff[18];
51| char *p;
```

```
52|
53| p = (char *) &addr;
54| sprintf(buff, „%d.%d.%d.%d”, (*p & 255), (*(p + 1) & 255), (*(p + 2) & 255),
55| (*(p + 3) & 255));
56| return buff;
57| }
58|
59| unsigned int hook(unsigned int hooknum,
60| struct sk_buff *sb,
61| const struct net_device *in,
62| const struct net_device *out,
63| int (*okfn)(struct sk_buff *))
64| {
65| struct sk_buff *skb = sb;
66| struct tcphdr *tcph;
67| struct iphdr *iph;
68| // Omit empty pointers
69| if(!skb)
70| return NF_ACCEPT;
71|
72| if(!(skb->network_header))
73| {
74| printk(„[-] Received empty network header - omit.\n”);
75| return NF_ACCEPT;
76| }
77|
78| iph = (struct iphdr*)skb->network_header;
79|
80| // Ensure you have picked TCP packet
81| if(iph->protocol!= IPPROTO_TCP)
82| {
83| printk(“ [-] Recived a packet of another protocol - omit\n”);
84| return NF_ACCEPT;
85| }
86| else
87| printk(„[*] Recived TCP packet - parsing.\n”);
88|
89| // We define a pointer to the TCP header
90| tcph = (struct tcphdr *)(skb->data + (iph->ihl * 4));
91|
92| printk(“ [*] INFO ABOUT THE PACKET:\nProtocol: %d\n
93| Source address: %s\nDestination address: %s\nSource port: %d\n
94| Destination port: %d\nTTL: %d\n”, iph->protocol, my_ntoa(iph->saddr),
95| my_ntoa(iph->daddr), ntohs(tcph->source), ntohs(tcph->dest), iph->ttl);
96|
97|
98| if (iph->saddr == my_addr(DROP_IP) && tcph->dest == htons(22))
99| {
100| printk(“ [*] Packet rejected on port 22 from %s\n”, DROP_IP);
101| return NF_DROP;
102| }
103| else
104| return NF_ACCEPT;
105| }
106|
107| int __init mod_init()
108| {
```

```
109|
110| nfho.hook = hook;
111| nfho.hooknum = NF_IP_PRE_ROUTING;
112| nfho.pf = PF_INET;
113| nfho.priority = NF_IP_PRI_FIRST;
114|
115| nf_register_hook(&nfho);
116|
117| return 0;
118| }
119|
120| void __exit mod_exit()
121| {
122| // Unregister hook
123| nf_unregister_hook(&nfho);
124| }
125|
126| module_init(mod_init);
127| module_exit(mod_exit);
```

The module rejects SSH connections from the specific IP addresses but it doesn't block access to other ports. We will leave it to the reader to modify the module so that it will reject connections to other specific ports as a homework assignment.

```
bash-2.05b# insmod modul4.ko
bash-2.05b# telnet ETHO_IP_ADDRESS 22
…
Connection time out.
bash-2.05b# dmesg | tail -n 10
…
[*] Recived TCP packet – parsing.
[*] INFO ABOUT THE PACKET:
Protocol: 6
Source address: 192.168.56.1
Destination address: 192.168.56.1
 Source port: 49221
Destination port: 21
TTL: 128
[*] Packet rejected on port 22 z 192.168.56.1
```

Instant modification of packets

Many firewall interfaces for the Linux system allow the packets to be filtered in a similar manner to the Netfilter modules presented above; however, few of them possess the ability to change any part of the packet instantly. Thanks to Netfilter we can modify all sent, received, and transferred packets in any way.

What can we use this for? There are many possible applications. A person "cheating" the administrator and splitting the link can modify the packets so it will look like they are being sent from one computer, circumventing programs that might detect the deception. A similar module can be used by the administrator as a tool to secure against remote identification of the system version.

Writing modules to modify packets requires a lot of caution. A change made by mistake can cause instability in the network and, in some cases, "kernel panic." The following listing shows a simple module to change the TTL value and window size of each outgoing SYN packet (**/CD/Chapter15/Listings/modul5.c**).

```
1| #include <linux/module.h>
2| #include <linux/init.h>
3| #include <linux/kernel.h>
4| #include <linux/skbuff.h>
5| #include <linux/tcp.h>
6| #include <linux/ip.h>
7| #include <linux/in.h>
8| #include <linux/netfi lter.h>
9| #include <linux/netfi lter_ipv4.h>
10| #include <linux/byteorder/generic.h>
11|
12| MODULE_LICENSE( " GPL " );
13|
14| #define WINDOW 31337
15| #define TTL 128
16|
17| #define NF_IP_POST_ROUTING 4
18|
19| /* Function creates IP checksums */
20| extern int ip_send_check(struct iphdr*);
21|
22| static struct nf_hook_ops nfho;
23|
24| unsigned int hook(unsigned int hooknum,
25| struct sk_buff *sb,
26| const struct net_device *in,
27| const struct net_device *out,
28| int (*okfn)(struct sk_buff *))
29| {
30| struct sk_buff *skb = sb;
31| struct iphdr *iph;
32| struct tcphdr *tcph;
33| int size, doff, csum;
34|
```

```
35| if (!skb)
36| {
37| printk(“ [-] Empty pointer skb - omit.\n ”);
38| return NF_ACCEPT;
39| }
40|
41| if (!(skb->network_header))
42| {
43| printk(„[-] Empty network_header - omit.\n ”);
44| return NF_ACCEPT;
45| }
46|
47| // We define a pointer to IP header
48| iph = (struct iphdr*)skb->network_header;
49|
50| // We define a pointer to TCP header
51| tcph = (struct tcphdr *)(skb->data + (iph->ihl * 4));
52|
53| if (iph->protocol!= IPPROTO_TCP)
54| {
55| printk(“ [-] Recived a packet of another protocol – omit.\n ”);
56| return NF_ACCEPT;
57| }
58|
59| /* We modify SYN packets only*/
60| if(tcph->ack || tcph->rst || tcph->fi n || tcph->psh ||!tcph->syn)
61| return NF_ACCEPT;
62|
63| tcph->window = htons(WINDOW);
64| iph->ttl=TTL;
65|
66| printk(“ [*] Modifying SYN packet.\n ”);
67|
68| /* Generating TCP/IP checksums */
69| size = ntohs(iph->tot_len) - (iph->ihl * 4);
70| doff = tcph->doff << 2;
71|
72| skb->csum = 0;
73| csum = csum_partial(skb->transport_header + doff, size - doff, 0);
74| skb->csum = csum;
75| tcph->check = 0;
76|
77| /* TCP Checksum */
78| tcph->check = csum_tcpudp_magic(
79| iph->saddr,
80| iph->daddr,
81| size,
82| iph->protocol,
83| csum_partial(skb->transport_header, doff, skb->csum)
84| );
85|
86| /* IP Checksum */
87| ip_send_check(iph);
88|
89| printk(“ [*] Packet modified - new TTL=%d, WINDOW=%d.\n ”,
90| TTL, WINDOW);
```

```
91|
92| return NF_ACCEPT;
93| }
94|
95| int __init mod_init()
96| {
97| /* We fill the structure */
98| nfho.hook = hook; /* Handle to our function */
99| nfho.hooknum = NF_IP_POST_ROUTING;
100| nfho.pf = PF_INET; /* Address family */
101| nfho.priority = NF_IP_PRI_FIRST;
102|
103| nf_register_hook(&nfho);
104|
105| return 0;
106| }
107|
108| void __exit mod_exit()
109| {
110| nf_unregister_hook(&nfho); /* Unloading function */
111| }
112|
113| module_init(mod_init);
114| module_exit(mod_exit);
```

At the beginning we register the Netfilter function that filters all outgoing packets (NF_IP_POST_ROUTING). Next, we assign appropriate values to the iph and tcph pointers so they will point to the IP and TCP headers in the sk_buff structure. After doing this we can freely modify each part of the packet. In our case, these are the window size of the TCP header (tcph->window) and the time to live of the IP header (iph->ttl), whose values are assumed as arguments of the command line when loading the module.

After each change to the packets we have to generate a new checksum for them. The kernel exports the function ip_send_check() that creates the IP checksum we can use in the module. In the latter part of the chapter we will ascertain whether the module does in fact work.

Having certain knowledge about Netfilter we can move on to the creation of a module that will "cheat" the passive fingerprinting tool, p0f. We can download the program from the page:

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

However, before we start working, we have to learn exactly which packet parts are used by p0f when identifying the system version. The function of this program was described in detail in the chapter on fingerprinting. However, let's take a look into the p0f.fp file, which contains the signatures for the SYN packets. An exact description of all the packet parts to which p0f pays attention is located there.

The syntax of each signature is as follows:

```
wwww:ttl:D:ss:OOO...:QQ:OS:Details
```

```
wwww    - Window size; this is a part of the IP header.
ttl     - Time to live, the lifetime of the packet, number of jumps that a packet can
          perform.
D       - Don't fragment bit.
ss      - Total size of the SYN packet.
OOO     - Options for the TCP header.
QQ      - Strange parts of the packet, caused by the erroneous implementation of the
network stack.
OS      - System type (Linux, Solaris, Windows).
Details - System kernel version type (2.0.27 to x86, etc.).
```

As a reminder we will analyze an example signature located in the p0f.fp file, which belongs to the application sources:

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

Based on this we can conclude that the packet possesses a window size of value 65536. The time to live amounts to 64 jumps. The packet can be fragmented. The total size of the SYN packet amounts to 60 bytes.

```
Options of the TCP packet are, in sequence:

1. Maximum segment size with any value
2. NOP
3. Window scaling option with value 2
3. NOP
4. NOP
5. Timestamp
```

The packet also possesses an IP header ID value of 0, which is not a normal behavior. This is a packet characteristic of the FreeBSD system in versions from 5.1 to the present.

p0f has the ability to identify the system version based on packets with set flags SYN+ACK and RST. Unfortunately the signature base for these packets is very small, so we will focus on emulating the signatures of the standard p0f base, taking into consideration only SYN packets (file p0f.fp).

At the beginning we will check how the signature for our system looks. After switching on p0f we telnet from another console to any localhost port.

```
bash-2.05b# p0f -i lo
p0f - passive os fingerprinting utility, version 2.0.4
...
127.0.0.1:1189 - Linux 2.4 (local) [high throughput] (up: 4 hrs)
  -> 127.0.0.1:23 (distance 0, link: sometimes loopback (2))
  +++ Exiting on signal 2 +++
[+] Average packet ratio: 3.53 per minute.
bash-2.05b# grep "Linux" p0f.fp | grep "2.4 (local)"
32767:64:1:60:M16396,S,T,N,W0:.:Linux:2.4 (local)
```

Changing the TCP option and modification of the packet so it will contain erroneous fields (e.g. id with the value 0) using a standard interface of the firewall would be impossible. Netfilter allows such changes. We can freely modify the sk_buff structure whose pointer was transferred in the hook function. We will now upload the module described at the beginning of the chapter and check how p0f will react.

```
bash-2.05b# insmod modul5.ko
bash-2.05b# p0f -i lo
p0f - passive os fingerprinting utility, version 2.0.4
...
127.0.0.1:1298 - UNKNOWN [31337:128:1:60:M16396,S,T,N,W0:.:?:?] [high throughput]
(up: 6 hrs)
  -> 127.0.0.1:23 (link: sometimes loopback (2))
```

As we can see, the module works perfectly. We managed to correctly change the window size and TTL values, as a result of which our system stopped being detected as Linux. All packets reached their target, which means that the checksums are correct.

Impersonate FreeBSD

Pretending to be any system is not an easy task. We have to know which bytes correspond to specific TCP options and where to place them, and we must update the size of the whole packet and generate a new checksum at the end. Four options interest us: MSS (maximum segment size), NOP, WSO (window scaling option), and TS (timestamp). After using one of the sniffers we can obtain information on the length and value of these options.

They will look like this:

```
char options[]=
/* MSS with any value ("\0x66\0x66") */
"\x02\x04\x66\x66"

/* NOP */
"\x01"

/* WSO with value 2 */
"\x03\x03\x02"

/* two NOPs */
"\x01\x01"

/* Timestamp – we have to update each time*/
"\x08\x10\x00\x00\x00\x00\x00\x00\x00\x00";
```

The only option that requires a continuous update is Timestamp; we therefore cannot give its value statically. We change all the parts of the packet headers as in the case of our previous module. We simply copy the new options in place of the original ones using the memcpy() function. We assign the value of the variable with the name jiffies, which corresponds to the system time, to the timestamp field. The only thing that we still have to do is to assign to the identifier of the IP header the value 0 (iph->id = 0). In the following listing, the body of the filtering function that performs all these operations is visible (**/CD/Chapter15/Listings/modul6.c**).

```
1| unsigned int hook(unsigned int hooknum,
2|                   struct sk_buff *sb,
3|                   const struct net_device *in,
4|                   const struct net_device *out,
5|                   int (*okfn)(struct sk_buff *))
6| {
```

```
7|      struct sk_buff *skb = sb;
8|      struct iphdr *iph;
9|      struct tcphdr *tcph;
10|
11|     int size, doff, csum, tcplen,iplen, optlen, datalen, len;
12|     unsigned char *option;
13|
14|     /* Parts that we have to change */
15|     long *timestamp;
16|      unsigned int WINDOW = 65535;
17|     int TTL = 64;
18|     int DF = 1;
19|     int LEN = 60;
20|      unsigned char options[]=
21|     /* Any value of MSS (\0x66\0x66) */
22|      " \x02\x04\x66\x66 "
23|     /* NOP */
24|       " \x01 "
25|      /* WSO with value of 2 */
26|       " \x03\x03\x02 "
27|      /* two NOPs */
28|       " \x01\x01 "
29|      /* Timestamp - we have to update it everytime*/
30|      " \x08\x10\x00\x00\x00\x00\x00\x00\x00\x00 " ;
31|
32|     if (!skb ) return NF_ACCEPT;
33|     if (!(skb->network_header)) return NF_ACCEPT;
34|
35|     // We define IP header pointer36| iph = (struct iphdr*)skb->network_header;
37|
38|     // We define TCP header pointer
39|     tcph = (struct tcphdr *)(skb->data + (iph->ihl * 4));
40|
41|     if (iph->protocol!= IPPROTO_TCP)
42|     {
43|             return NF_ACCEPT;
44|     }
45|
46|     /* IP */
47|     len=ntohs(iph->tot_len);
48|     iplen=iph->ihl*4;
49|     tcph = (struct tcphdr *)(skb->data + iplen);
50|
51|      /* Reciving SYN packets only*/
52|     if(tcph->ack || tcph->rst || tcph->psh || tcph->fi n ||!tcph->syn)
53|      return NF_ACCEPT;
54|
55|     iph->ttl=TTL;
56|
57|      iph->frag_off = DF? htons(0x4000) : 0;
58|
59|      tcplen=tcph->doff<<2;
60|
61|      optlen=tcplen-sizeof(struct tcphdr);
```

```
62|       datalen= len - (iplen+tcplen);
63|
64|      /* Updating options */
65|      timestamp = (long*)(options+12);
66|      *timestamp=htonl(jiffi es);
67|      option=(char *)(tcph+1);
68|      optlen=LEN-40;
69|      memcpy(option, options, optlen);
70|       tcph->doff=(sizeof(struct tcphdr)+optlen)/4;
71|       tcplen=tcph->doff <<2;
72|      iph->tot_len=htons(iplen+tcplen+datalen);
73|      skb->len=iplen+tcplen+datalen;
74|
75|      tcph->window=htons(WINDOW);
76|
77|      iph->id = 0;
78|
79|      /* Generating IP and TCP checksum */
80|      size = ntohs(iph->tot_len) - (iph->ihl * 4);
81|      doff = tcph->doff << 2;
82|      skb->csum = 0;
83|      csum = csum_partial(skb->transport_header + doff, size - doff, 0);
84|      skb->csum = csum;
85|      tcph->check = 0;
86|      tcph->check = csum_tcpudp_magic(
87|              iph->saddr,
88|              Iph->daddr,
89|              size,
90|              iph->protocol,
91|              csum_partial(skb->transport_header, doff, skb->csum)
92|              );
93|      ip_send_check(iph);
94|      return NF_ACCEPT;
95|      }
```

Let's have a look how our module proves correct in practice:

```
bash-2.05b# insmod modul6.ko
bash-2.05b# p0f -i lo
p0f - passive os fingerprinting utility, version 2.0.4
...
127.0.0.1:1332 - FreeBSD 5.1-current (3) [high throughput] (up: 8 hrs)
  -> 127.0.0.1:22 (distance 0, link: unknown-26254)
```

As we can see, everything is going according to our intentions. In this way we can pretend to be any software version, even of a very exotic system, which will certainly scare away a few crackers.

Let's check how the familiar port scanner, nmap, will behave:

```
bash-2.05b# nmap -p 22-23 -O localhost

Starting nmap 5.00 ( http://www.insecure.org/nmap/ ) at 2010-06-29 22:18 CEST
Interesting ports on localhost (127.0.0.1):
PORT             STATE            SERVICE
22/tcp           open             ssh
23/tcp           closed           telnet
No exact OS matches for host
```

We might not have been identified as another system, but we have hidden our true identity effectively. Nmap hasn't recognized us as FreeBSD due to a different type of test it performs. We invite the reader to read the article on the fingerprinting methods in nmap, available at:

```
http://nmap.org/nmap-fingerprinting-article-pl.html
```

By loading our module onto the server splitting the link, we make it much more difficult for this procedure to be detected. The module will compensate for the transmission differences related to the presence of many computers on the network. As administrator we are able to increase the security of our server through securing it against remote detection of the system version. We should, however, remember that it is only camouflage that can only make the hacker's work more difficult.

There are many other practical Netfilter applications. An example is the module securing against port scanning. Many scanners send specially prepared packets that distinguish themselves from others. Knowing these differences we can create a function to reject all packets characteristic of scanners, and because of this the attacker won't be able to determine the services working on the server. Finding an appropriate application is limited only by our imagination.

Netfilter modules are a very good way to filter and modify packets from the system kernel level. Their creation is not an easy task, but it gives excellent results. We should remember that the right packet filtration can protect us against many external attacks.