1

CARE AND FEEDING OF IPTABLES

In this chapter we'll explore essential aspects of properly installing, maintaining, and interacting with the iptables firewall on Linux systems. We'll cover iptables administration from the perspectives of both kernel and userland, as well as how to build and maintain an iptables firewall policy. A default policy will be constructed that will serve as a guide throughout several chapters in the book; a script that implements it and a network diagram are included for reference in this chapter. Many of the example attacks throughout this book will be launched from hosts shown in this network diagram. Finally, we'll cover testing the default iptables policy to ensure that it is functioning as designed.

iptables

The iptables firewall is developed by the Netfilter Project (http://www.netfilter.org) and has been available to the masses as part of Linux since the release of the Linux 2.4 kernel in January 2001.

Over the years, iptables has matured into a formidable firewall with most of the functionality typically found in proprietary commercial firewalls. For example, iptables offers comprehensive protocol state tracking, packet application layer inspection, rate limiting, and a powerful mechanism to specify a filtering policy. All major Linux distributions include iptables, and many prompt the user to deploy an iptables policy right from the installer.

The differences between the terms *iptables* and *Netfilter* have been a source of some confusion in the Linux community. The official project name for all of the packet filtering and mangling facilities provided by Linux is *Netfilter*, but this term also refers to a framework within the Linux kernel that can be used to hook functions into the networking stack at various stages. On the other hand, *iptables* uses the Netfilter framework to hook functions designed to perform operations on packets (such as filtering) into the networking stack. You can think of Netfilter as providing the framework on which iptables builds firewall functionality.

The term *iptables* also refers to the userland tool that parses the command line and communicates a firewall policy to the kernel. Terms such as *tables*, *chains*, *matches*, and *targets* (defined later in this chapter) make sense in the context of iptables.

Netfilter does not filter traffic itself—it just allows functions that *can* filter traffic to be hooked into the right spot within the kernel. (I will not belabor this point; much of the material in this book centers around iptables and how it can take action against packets that match certain criteria.) The Netfilter Project also provides several pieces of infrastructure in the kernel, such as connection tracking and logging; any iptables policy can use these facilities to perform specialized packet processing.

NOTE *In this book I will refer to log messages generated by the Netfilter logging subsystem as* iptables log messages*; after all, packets are only logged upon matching a* `LOG` *rule that is constructed by iptables in the first place. So as to not confuse things, I will use the term* iptables *by default unless there is a compelling reason to use* Netfilter *(such as when discussing kernel compilation options or connection-tracking capabilities). Most people associate Linux firewalls with iptables, anyway.*

Packet Filtering with iptables

The iptables firewall allows the user to instrument a high degree of control over IP packets that interact with a Linux system; that control is implemented within the Linux kernel. A policy can be constructed with iptables that acts as a vigorous traffic cop—packets that are not permitted to pass fall into oblivion and are never heard from again, whereas packets that pass muster are sent on their merry way or altered so that they conform to local network requirements.

An iptables policy is built from an ordered set of *rules*, which describe to the kernel the actions that should be taken against certain classes of packets. Each iptables rule is applied to a chain within a table. An iptables *chain* is a

collection of rules that are compared, in order, against packets that share a common characteristic (such as being routed to the Linux system, as opposed to away from it).

Tables

A *table* is an iptables construct that delineates broad categories of functionality, such as packet filtering or Network Address Translation (NAT). There are four tables: `filter`, `nat`, `mangle`, and `raw`. Filtering rules are applied to the `filter` table, NAT rules are applied to the `nat` table, specialized rules that alter packet data are applied to the `mangle` table, and rules that should function independently of the Netfilter connection-tracking subsystem are applied to the `raw` table.

Chains

Each table has its own set of built-in chains, but user-defined chains can also be created so that the user can build a set of rules that is related by a common tag such as `INPUT_ESTABLISHED` or `DMZ_NETWORK`. The most important built-in chains for our purposes are the `INPUT`, `OUTPUT`, and `FORWARD` chains in the `filter` table:

- The `INPUT` chain is traversed by packets that are destined for the local Linux system after a routing calculation is made within the kernel (i.e., packets destined for a local socket).
- The `OUTPUT` chain is reserved for packets that are generated by the Linux system itself.
- The `FORWARD` chain governs packets that are routed through the Linux system (i.e., when the iptables firewall is used to connect one network to another and packets between the two networks must flow through the firewall).

Two additional chains that are important for any serious iptables deployment are the `PREROUTING` and `POSTROUTING` chains in the `nat` table, which are used to modify packet headers before and after an IP routing calculation is made within the kernel. Sample iptables commands illustrate the usage of the `PREROUTING` and `POSTROUTING` chains later in this chapter, but in the meantime, Figure 1-1 shows how packets flow through the `nat` and `filter` tables within the kernel.

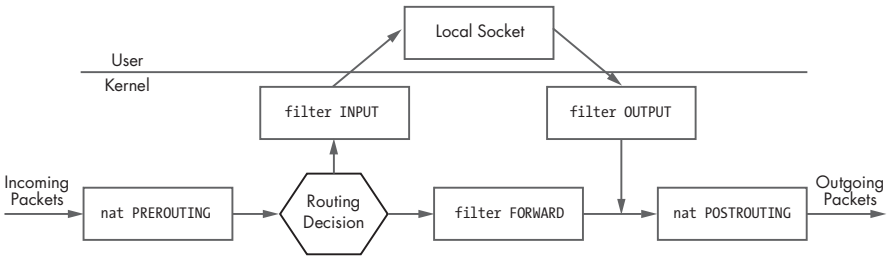

Figure 1-1: iptables packet flow

Matches

Every iptables rule has a set of matches along with a *target* that tells iptables what to do with a packet that conforms to the rule. An iptables *match* is a condition that must be met by a packet in order for iptables to process the packet according to the action specified by the rule target. For example, to apply a rule only to TCP packets, you can use the `--protocol` match.

Each match is specified on the iptables command line. The most important iptables matches for this book are listed below. (You'll see more about matches in "Default iptables Policy" on page 20 when we discuss the default iptables policy used throughout this book.)

`--source (-s)`	Match on a source IP address or network
`--destination (-d)`	Match on a destination IP address or network
`--protocol (-p)`	Match on an IP value
`--in-interface (-i)`	Input interface (e.g., eth0)
`--out-interface (-o)`	Output interface
`--state`	Match on a set of connection states
`--string`	Match on a sequence of application layer data bytes
`--comment`	Associate up to 256 bytes of comment data with a rule within kernel memory

Targets

Finally, iptables supports a set of targets that trigger an action when a packet matches a rule.[1] The most important targets used in this book are as follows:

`ACCEPT`	Allows a packet to continue on its way.
`DROP`	Drops a packet. No further processing is performed, and as far as the receiving stack is concerned, it is as though the packet was never sent.
`LOG`	Logs a packet to syslog.
`REJECT`	Drops a packet and simultaneously sends an appropriate response packet (e.g., a TCP Reset packet for a TCP connection or an ICMP Port Unreachable message for a UDP packet).
`RETURN`	Continues processing a packet within the calling chain.

We'll build ample iptables rules that use several of the matches and targets discussed above in "Default iptables Policy" on page 20.

Installing iptables

Because iptables is split into two fundamental components (kernel modules and the userland administration program), installing iptables involves compiling and installing both the Linux kernel and the userland binary. The

[1] Note that *matching* here is used to mean that a packet conforms to all of the match criteria contained within an iptables rule.

kernel source code contains many Netfilter subsystems, and the essential packet-filtering capability is enabled by default in the pristine authoritative kernels released on the official Linux Kernel Archives website, http://www.kernel.org.

In some of the earlier 2.6 kernels (and all of the 2.4 kernels), the Netfilter compilation options were not enabled by default. However, because the software provided by the Netfilter Project has achieved a high level of quality over the years, the kernel maintainers felt it had reached a point where using iptables on Linux should not require you to recompile the kernel. Recent kernels allow you to filter packets by default with an iptables policy.

While many Linux distributions come with pre-built kernels that already have iptables compiled in, the default kernel configuration in a kernel downloaded from http://www.kernel.org tries to stay as lean and mean as possible out of the box, so not all Netfilter subsystems may be enabled. For example, the Netfilter connection-tracking capability is not enabled by default in the 2.6.20.1 kernel (the most recent kernel version as of this writing). Hence, it is important to understand the process of recompiling the kernel so that iptables policies can make use of additional functionality.

NOTE *Throughout this chapter, some of the compilation output and installation commands have been abbreviated to save space and keep the focus on what is important.*

The most important step towards building a Linux system that can function as an iptables firewall is the proper configuration and compilation of the Linux kernel. All heavy network-processing and comparison functions in iptables take place within the kernel, and we'll begin by compiling the latest version of the kernel from the 2.6 stable series. Although a complete treatment of the vagaries of the kernel compilation process is beyond the scope of this book, we'll discuss enough of the process for you to compile in and enable the critical capabilities of packet filtering, connection tracking, and logging. As far as other kernel compilation options not related to Netfilter subsystems, such as processor architecture, network interface driver(s), and filesystem support, I'll assume that you've chosen the correct options such that the resulting kernel will function correctly on the hardware on which it is deployed.

NOTE *For more information on compiling the 2.6 series kernel, see the Kernel Rebuild Guide written by Kwan Lowe (http://www.digitalhermit.com/~kwan/kernel.html). For the older 2.4 kernels, see the Kernel-HOWTO written by Brian Ward (http://www.tldp.org/HOWTO/Kernel-HOWTO.html), or refer to any good book on Linux system administration. Brian Ward's* How Linux Works *(No Starch Press, 2004) also covers kernel compilation.*

Before you can install the Linux kernel, you need to download and unpack it. The following commands accomplish this for the 2.6.20.1 kernel. (In these commands, I assume the directory /usr/src is writable by the current user.)

NOTE *Except where otherwise noted, this chapter is written from the perspective of the 2.6-series kernel because it represents the latest and greatest progeny of the Linux kernel developers. In general, however, the same strategies also apply to the 2.4-series kernel.*

```
$ /usr/src
$ wget http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.20.1.tar.bz2
$ tar xfj linux-2.6.20.1.tar.bz2
$ ls -ld linux-2.6.20.1
drwxr-xr-x  18 mbr users 600 Jun 16 20:48 linux-2.6.20.1
```

Although I have chosen specific kernel versions in the commands above, the analogous commands apply for newer kernel versions. For example when, say, the 2.6.20.2 kernel is released, you only need to substitute `2.6.20.1` with **`2.6.20.2`** in the above commands.

NOTE *One thing to keep in mind is that the load on the kernel.org webserver has been steadily increasing over the years, and a random glance at the bandwidth utilization graphs on http://www.kernel.org shows the current utilization at well over 300 Mbps. To help reduce the load, the kernel can be downloaded from one of the mirrors listed at http://www.kernel.org/mirrors. Once you have a particular version of the kernel sources on your system, you can download and apply a kernel patch file to upgrade to the next version. (The patch files are much smaller than the kernel itself.)*

Kernel Configuration

Before you can begin compiling, you must construct a kernel configuration file. Fortunately, the process of building this file has been automated by kernel developers, and it can be initiated with a single command (within the /usr/src/linux-2.6.20.1 directory):

```
$ make menuconfig
```

The `make menuconfig` command launches the Ncurses interface in which you can select various compile options. (You can call the X Windows or terminal interface with the commands `make xconfig` and `make config`, respectively.) I've chosen the Ncurses interface because it provides a nice balance between the spartan terminal interface and the relatively expensive X Windows interface. The Ncurses interface also easily lends itself to the configuration of a remote Linux kernel across an SSH session without having to forward an X Windows connection.

After executing `make menuconfig`, we are presented with several configuration sections ranging from Code Maturity Level options to Library Routines. Most Netfilter compilation options for the 2.6-series kernel are located within a section called Network Packet Filtering Framework (Netfilter) under Networking ▸ Networking Options.

Essential Netfilter Compilation Options

Some of the more important options to enable within the kernel configuration file include Netfilter connection tracking, logging, and packet filtering. (Recall that iptables builds a policy by using the in-kernel framework provided by Netfilter.)

There are two additional configuration sections in the Network Packet Filtering Framework (Netfilter) section—Core Netfilter Configuration and IP: Netfilter Configuration.

Core Netfilter Configuration

The Core Netfilter Configuration section contains several important options that should all be enabled:

- Comment match support
- FTP support
- Length match support
- Limit match support
- MAC address match support
- `MARK` target support
- Netfilter connection tracking support
- Netfilter `LOG` over NFNETLINK interface
- Netfilter netlink interface
- Netfilter Xtables support
- State match support
- String match support

IP: Netfilter Configuration

With the Core Netfilter Configuration section completed, we'll move on to the IP: Netfilter Configuration section. The options that should be enabled within this section are as follows:

- ECN target support
- Full NAT
- IP address range match support
- IP tables support (required for filtering/masq/NAT)
- IPv4 connection tracking support (required for NAT)
- `LOG` target support
- `MASQUERADE` target support
- Owner match support
- Packet filtering
- Packet mangling

- raw table support (required for NOTRACK/TRACE)
- Recent match support
- REJECT target support
- TOS match support
- TOS target support
- TTL match support
- TTL target support
- ULOG target support

In the 2.6 kernel series, the individual compilation sections underwent a major reorganization. In the older 2.4 series, the IP: Netfilter Configuration section can be found underneath Networking Options, and this section is only visible if the Network Packet Filtering option is enabled.

Finishing the Kernel Configuration

Having configured the 2.6.20.1 kernel with the required Netfilter support via the menuconfig interface, save the kernel configuration file by selecting **Exit** until you see the message *Do you wish to save your new kernel configuration?* Answer **Yes**.

After saving the new kernel configuration, you are dropped back to the command shell where you can examine the resulting Netfilter compilation options via the following commands.

NOTE *The output of these commands is too long to include here, but most Netfilter options, such as CONFIG_IP_NF_NAT and CONFIG_NETFILTER_XT_MATCH_STRING, for example, contain either the substring _NF_ or the substring NETFILTER.*

```
$ grep "_NF_" .config
$ grep NETFILTER .config
```

Loadable Kernel Modules vs. Built-in Compilation and Security

Most of the Netfilter subsystems enabled in the previous section may be compiled either as a Loadable Kernel Module (LKM), which can be dynamically loaded or unloaded into or out of the kernel at run time, or compiled directly into the kernel, in which case they cannot be loaded or unloaded at run time. In the configuration section above, we have chosen to compile most Netfilter subsystems as LKMs.

There is a security trade-off between compiling functionality as an LKM and compiling directly into the kernel. On one hand, any feature that is compiled as an LKM can be removed from a running kernel with the rmmod command. This can provide an advantage if a security vulnerability is discovered within the module, because in some cases the vulnerability can be mitigated just by unloading the module. Too, if the vulnerability has been patched in the kernel sources, the module can be recompiled and redeployed without ever taking the system down completely; fixing the vulnerability would involve zero downtime.

NOTE *Netfilter subsystems in the kernel are not immune from the occasional security vulnerability. For example, a vulnerability was discovered in the code that handles TCP options in the Netfilter logging subsystem (see http://www.netfilter.org/security/2004-06-30-2.6-tcpoption.html). If the logging subsystem was compiled as a module, the kernel can be protected by sacrificing the ability of iptables to create log messages by unloading the module, which seems like a good trade-off.*

On the other hand, if a vulnerability is discovered within the code that implements a feature and this code is compiled directly into the kernel, the only way to fix the vulnerability is to apply a patch, recompile, and then reboot the entire system into the new (fixed) kernel. For mission-critical systems (such as a corporate DNS server), this may not be feasible until an outage window can be scheduled, and in the meantime the system may be vulnerable to a kernel-level compromise.

ROOTKIT THREAT

The story does not end here, however. Compiling a kernel with loadable module support opens up a sinister possibility: If an attacker successfully compromises the system, having module support in the kernel makes it easier for the attacker to install a kernel-level rootkit. Once the kernel itself is compromised, all sorts of mischief can be levied against the system.

Compromising the kernel itself represents the crown jewel of all compromises; filesystem integrity checkers such as Tripwire can be fooled, processes can be hidden, and network connections can be shielded from the view of tools like netstat and lsof, and even from packet sniffers (executed locally). Simply compiling the kernel without module support is not a foolproof solution, however, since not all kernel-level rootkits require the host kernel to offer module support. For example, the SucKIT rootkit can load itself into a running kernel by directly manipulating kernel memory through the /dev/kmem character device.* The SucKIT rootkit was introduced to the security community in the *Phrack* magazine article "Linux on-the-fly kernel patching without LKM" (see http://www.phrack.org).

*A *character device* is an interface to the kernel that can be accessed as a stream of bytes instead of just by discrete block sizes, as in the case of a block device. Examples of character devices include /dev/console and the serial port device files, such as /dev/ttyS0.

The power of module loading and unloading provides a degree of flexibility that is attractive, so this is the strategy I chose here. When making your own choice, be sure to consider the trade-offs.

Security and Minimal Compilation

Regardless of the strategy you choose for compiling Netfilter subsystems—whether as LKM's or directly into the kernel—an overriding fact in computer security is that complexity breeds insecurity; more complex systems are harder to secure. Fortunately, iptables is highly configurable both in terms of the run-time rules language used to describe how to process and filter network traffic and also in terms of the categories of supported features controlled by the kernel compilation options.

To reduce the complexity of the code running in the kernel, do not compile features that you don't need. Removing unnecessary code from a running kernel helps to minimize the risks from as yet undiscovered vulnerabilities lurking in the code.

For example, if you have no need for logging support, simply do not enable the Log Target Support option in the menuconfig interface. If you have no need for the stateful tracking of FTP connections, leave the FTP Protocol Support option disabled. If you do not need to be able to write filter rules against MAC addresses in Ethernet headers, disable the MAC Address Match Support option.

Only compile in the features that are absolutely necessary to meet the networking and security needs of the local network and/or host.

Kernel Compilation and Installation

Now that our kernel is configured, we'll move on to the compilation and installation. As previously mentioned, we assume that all other necessary kernel options (such as processor architecture) have been selected for the proper support of the hardware on which the new kernel will run.

To compile and install the new 2.6.20.1 kernel within the /boot partition, execute the following commands:

```
$ make
$ su -
Password:
# mount /boot
# cd /usr/src/linux-2.6.20.1
# make install && make modules_install
```

The successful conclusion of the above commands heralds the need to configure the bootloader and finally to boot into the new 2.6.20.1 kernel. Assuming that you're using the GRUB bootloader and that the mount point for the root partition is /dev/hda2, add the following lines to the /boot/grub/grub.conf file using your favorite editor:

```
title  linux-2.6.20.1
root (hd0,0)
kernel /boot/vmlinuz-2.6.20.1 root=/dev/hda2
```

Now, reboot!

```
# shutdown -r now
```

Installing the iptables Userland Binaries

Having installed and booted into a kernel that has Netfilter hooks compiled in, we'll now install the latest version of the iptables userland program. To do so, first download and unpack the latest iptables sources in the /usr/local/src directory, and then check the MD5 sum[2] against the published value at http://www .netfilter.org:

```
$ cd /usr/local/src/
$ wget http://www.netfilter.org/projects/iptables/files/iptables-1.3.7.tar.bz2
$ md5sum 1.3.7.tar.bz2
dd965bdacbb86ce2a6498829fddda6b7  iptables-1.3.7.tar.bz2
$ tar xfj iptables-1.3.7.tar.bz2
$ cd iptables-1.3.7
```

For the compilation and installation steps of the iptables binary, recall that we compiled the kernel within the directory /usr/src/linux-2.6.20.1; compiling iptables requires access to the kernel source code because it compiles against C header files in directories such as include/linux/netfilter_ipv4 in the kernel source tree. We'll use the /usr/src/linux-2.6.20.1 directory to define the `KERNEL_DIR` variable on the command line, and the `BINDIR` and `LIBDIR` variables allow us to control the paths where the iptables binary and libraries are installed. You can compile and install iptables as follows:

```
$ make KERNEL_DIR=/usr/src/linux-2.6.20.1 BINDIR=/sbin LIBDIR=/lib
$ su -
Password:
# cd /usr/local/src/iptables-1.3.7
# make install KERNEL_DIR=/usr/src/linux-2.6.20.1 BINDIR=/sbin LIBDIR=/lib
```

For the final proof that we have installed iptables and that it can interact with the running 2.6.20.1 kernel, we'll issue commands to display the iptables version number and then instruct it to list the current ruleset in the `INPUT`, `OUTPUT`, and `FORWARD` chains (which at this point contain no active rules):

```
# which iptables
/sbin/iptables
# iptables -V
iptables v1.3.7
# iptables -nL
Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
```

[2] You should also check the digital signature made with GnuPG against the published value at http://www.netfilter.org. This requires importing the Netfilter GnuPG public key, and running the `gpg --verify` command against the signature file. Details of this process for the psad project can be found in Chapter 5, and similar steps apply here to the iptables-1.3.7 tarball.

NOTE *Most Linux distributions already have iptables installed, so you may not need to go through the installation process above. However, to ensure you have a system that is prepared for the discussion in this book, it may be a good idea to have the latest version of iptables installed. As you will see in Chapter 9, the string matching capability is critical for running fwsnort, so you may need to upgrade your kernel if it doesn't already support this (see "Kernel Configuration" on page 14).*

Default iptables Policy

We now have a functioning Linux system with iptables installed. The remainder of this chapter will concentrate on various administrative and run-time aspects of iptables firewalls.

We'll begin by constructing a Bourne shell script (iptables.sh) to implement an iptables filtering policy tailored for a modest network with a permanent Internet connection. This policy will be used throughout the rest of the book and serves as a common ground—we will refer to this policy in several subsequent chapters. You can also download the iptables.sh script from http://www.cipherdyne.org/LinuxFirewalls. But first, here is some background information on iptables.

Policy Requirements

Let's define the requirements for an effective firewall configuration for a network consisting of several client machines and two servers. The servers (a webserver and a DNS server) must be accessible from the external network. Systems on the internal network should be allowed to initiate the following types of traffic through the firewall to external servers:

- Domain Name System (DNS) queries
- File Transfer Protocol (FTP) transfers
- Network Time Protocol (NTP) queries
- Secure SHell (SSH) sessions
- Simple Mail Transfer Protocol (SMTP) sessions
- Web sessions over HTTP/HTTPS
- whois queries

Except for access to the services listed above, all other traffic should be blocked. Sessions initiated from the internal network or directly from the firewall should be statefully tracked by iptables (with packets that do not conform to a valid state logged and dropped as early as possible), and NAT services should also be provided.

In addition, the firewall should also implement controls against spoofed packets from the internal network being forwarded to any external IP address:

- The firewall itself must be accessible via SSH from the internal network, but from nowhere else unless it is running fwknop for authentication

(covered in Chapter 13); SSH should be the only server process running on the firewall.

- The firewall should accept ICMP Echo Requests from both the internal and external networks, but unsolicited ICMP packets that are not Echo Requests should be dropped from any source IP address.
- Lastly, the firewall should be configured with a default *log and drop stance* so that any stray packets, port scans, or other connection attempts that are not explicitly allowed through will be logged and dropped.

NOTE *We'll assume that the external IP address on the firewall is statically assigned by the ISP, but a dynamically assigned IP address would also work because we restrict packets on the external network by interface name on the firewall instead of by IP address.*

To simplify the task of building the iptables policy, assume there is a single internal network with a non-routable network address of 192.168.10.0[3] and a Class C subnet mask 255.255.255.0 (or /24 in CIDR notation).

The internal network interface on the firewall (see Figure 1-2) is eth1 with IP address 192.168.10.1, and all internal hosts have this address as their default gateway. This allows internal systems to route all packets destined for systems that are not within the 192.168.10.0/24 subnet out through the firewall. The external interface on the firewall is eth0, and so as to remain network agnostic, we designate an external IP address of 71.157.*X.X* to this interface.

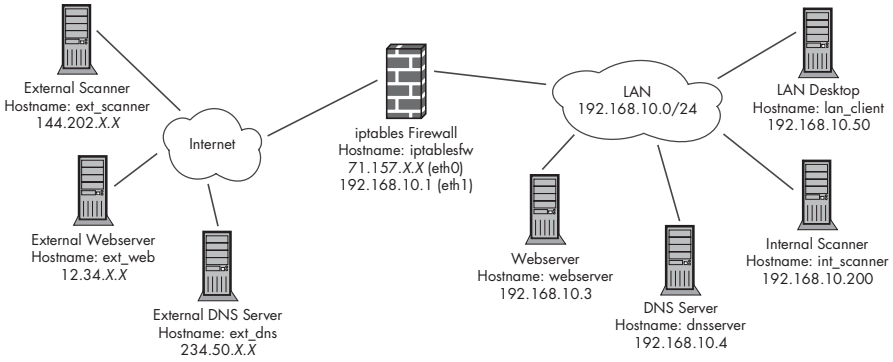

Figure 1-2: Default network diagram

There are two malicious systems represented: one on the internal network (192.168.10.200, hostname int_scanner) and the other on the external network (144.202.*X.X*, hostname ext_scanner). The network diagram in Figure 1-2 is included for reference here, and we will refer to it in later chapters as well. All traffic examples in the book reference the network diagram in Figure 1-2 unless otherwise noted, and you will see the hostnames in this diagram used at the shell prompts where commands are executed so that it is clear which system is generating or receiving traffic.

[3] The set of all non-routable addresses is defined in RFC 1918. Such addresses are non-routable by convention on the open Internet.

iptables.sh Script Preamble

To begin the iptables.sh script, it is useful to define three variables, `IPTABLES` and `MODPROBE` (for the paths to the iptables and modprobe binaries) and `INT_NET` (for the internal subnet address and mask), that will be used throughout the script (see ❶ below). At ❷ any existing iptables rules are removed from the running kernel, and the filtering policy is set to `DROP` on the `INPUT`, `OUTPUT`, and `FORWARD` chains. Also, the connection-tracking modules are loaded with the `modprobe` command.

```
[iptablesfw]# cat iptables.sh
#!/bin/sh
❶ IPTABLES=/sbin/iptables
MODPROBE=/sbin/modprobe
INT_NET=192.168.10.0/24

### flush existing rules and set chain policy setting to DROP
echo "[+] Flushing existing iptables rules..."
❷ $IPTABLES -F
$IPTABLES -F -t nat
$IPTABLES -X
$IPTABLES -P INPUT DROP
$IPTABLES -P OUTPUT DROP
$IPTABLES -P FORWARD DROP
### load connection-tracking modules
$MODPROBE ip_conntrack
$MODPROBE iptable_nat
$MODPROBE ip_conntrack_ftp
$MODPROBE ip_nat_ftp
```

The INPUT Chain

The `INPUT` chain is the iptables construct that governs whether packets that are destined for the local system (that is, after the result of a routing calculation made by the kernel designates that the packet is destined for a local IP address) may talk to a local socket. If the first rule in the `INPUT` chain instructs iptables to drop all packets (or if the policy setting of the `INPUT` chain is set to `DROP`), then all efforts to communicate directly with the system over any IP communications (such as TCP, UDP, or ICMP) will fail. The *Address Resolution Protocol (ARP)* is also an important class of traffic that is ubiquitous on Ethernet networks. However, because ARP works at the data link layer instead of the network layer, iptables cannot filter such traffic, since it only filters IP traffic and overlying protocols.

Hence, ARP requests and replies are sent and received regardless of the iptables policy. (It is possible to filter ARP traffic with arptables, but a discussion of this topic is beyond the scope of this book, since we generally concentrate on the network layer and above.)

NOTE *iptables can filter IP packets based on data link layer MAC addresses, but only if the kernel is compiled with the MAC address extension enabled. In the 2.4 kernel series, the MAC address extension must be manually enabled, but the 2.6 kernel series enables it by default.*

Continuing with the development of the iptables shell script, after the preamble, we use the following commands to set up the INPUT chain.

```
###### INPUT chain ######
echo "[+] Setting up INPUT chain..."
### state tracking rules
❸ $IPTABLES -A INPUT -m state --state INVALID -j LOG --log-prefix "DROP INVALID "
--log-ip-options --log-tcp-options
$IPTABLES -A INPUT -m state --state INVALID -j DROP
$IPTABLES -A INPUT -m state --state ESTABLISHED,RELATED -j ACCEPT

### anti-spoofing rules
❹ $IPTABLES -A INPUT -i eth1 -s ! $INT_NET -j LOG --log-prefix "SPOOFED PKT "
$IPTABLES -A INPUT -i eth1 -s ! $INT_NET -j DROP

### ACCEPT rules
❺ $IPTABLES -A INPUT -i eth1 -p tcp -s $INT_NET --dport 22 --syn -m state
--state NEW -j ACCEPT
$IPTABLES -A INPUT -p icmp --icmp-type echo-request -j ACCEPT

### default INPUT LOG rule
❻ $IPTABLES -A INPUT -i ! lo -j LOG --log-prefix "DROP " --log-ip-options
--log-tcp-options
```

Recall that our firewall policy requirements mandate that iptables statefully tracks connections; packets that do not match a valid state should be logged and dropped early. This is accomplished by the three iptables commands beginning at ❸ above; you will see a similar set of three commands for the OUTPUT and FORWARD chains as well. The state match is used by each of these rules, along with the criteria of INVALID, ESTABLISHED, or RELATED. The INVALID state applies to packets that cannot be identified as belonging to any existing connection—for example, a TCP FIN packet that arrives out of the blue (i.e., when it is not part of any TCP session) would match the INVALID state. The ESTABLISHED state triggers on packets only after the Netfilter connection-tracking subsystem has seen packets in both directions (such as acknowledgment packets in a TCP connection through which data is being exchanged). The RELATED state describes packets that are starting a new connection[4] in the Netfilter connection-tracking subsystem, but this connection is associated with an existing one—for example, an ICMP Port Unreachable message that is returned after a packet is sent to a UDP socket where no server is bound. Next, anti-spoofing rules are added at ❹ so packets that originate from the internal network must have a source address within the 192.168.10.0/24 subnet. At ❺ are two ACCEPT rules for SSH connections from the internal network, and ICMP Echo Requests are accepted from any source. The rule that accepts SSH connections uses the state match with a state of NEW together with the iptables --syn command-line argument. This only matches on TCP packets with FIN, RST, and ACK flags zeroed-out and the SYN flag set, and then only if the NEW state is matched (which means that the packet is starting a new connection, as far as the connection-tracking subsystem is concerned).

[4] Here *connection* is the tracking mechanism that Netfilter uses to categorize packets.

Finally at ❻ is the default LOG rule.[5] Recall from the script preamble that packets that are not accepted by some rule within the INPUT chain will be dropped by the DROP policy assigned to the chain; this also applies to the OUTPUT and FORWARD chains. As you can see, the configuration of the INPUT chain is exceedingly easy, since we only need to accept incoming connection requests to the SSH daemon from the internal network, enable state tracking for locally generated network traffic, and finally log and drop unwanted packets (including spoofed packets from the internal network). Similar configurations apply to OUTPUT and FORWARD chains, as you'll see below.

The OUTPUT Chain

The OUTPUT chain allows iptables to apply kernel-level controls to network packets generated by the local system. For example, if an SSH session is initiated to an external system by a local user, the OUTPUT chain could be used to either permit or deny the outbound SYN packet.

The commands in the iptables.sh script that build the OUTPUT chain ruleset appear below:

```
###### OUTPUT chain ######
echo "[+] Setting up OUTPUT chain..."
### state tracking rules
$IPTABLES -A OUTPUT -m state --state INVALID -j LOG --log-prefix "DROP
INVALID " --log-ip-options --log-tcp-options
$IPTABLES -A OUTPUT -m state --state INVALID -j DROP
$IPTABLES -A OUTPUT -m state --state ESTABLISHED,RELATED -j ACCEPT

### ACCEPT rules for allowing connections out
❼ $IPTABLES -A OUTPUT -p tcp --dport 21 --syn -m state --state NEW -j ACCEPT
$IPTABLES -A OUTPUT -p tcp --dport 22 --syn -m state --state NEW -j ACCEPT
$IPTABLES -A OUTPUT -p tcp --dport 25 --syn -m state --state NEW -j ACCEPT
$IPTABLES -A OUTPUT -p tcp --dport 43 --syn -m state --state NEW -j ACCEPT
$IPTABLES -A OUTPUT -p tcp --dport 80 --syn -m state --state NEW -j ACCEPT
$IPTABLES -A OUTPUT -p tcp --dport 443 --syn -m state --state NEW -j ACCEPT
$IPTABLES -A OUTPUT -p tcp --dport 4321 --syn -m state --state NEW -j ACCEPT
$IPTABLES -A OUTPUT -p udp --dport 53 -m state --state NEW -j ACCEPT
$IPTABLES -A OUTPUT -p icmp --icmp-type echo-request -j ACCEPT

### default OUTPUT LOG rule
$IPTABLES -A OUTPUT -o ! lo -j LOG --log-prefix "DROP " --log-ip-options
--log-tcp-options
```

In accordance with our policy requirements, at ❼ we'll assume that connections initiated from the firewall itself will be to download patches or software over FTP, HTTP, or HTTPS; to initiate outbound SSH and SMTP connections; or to issue DNS or whois queries against other systems.

[5] One thing to note about the iptables.sh script is that all of the LOG rules are built with the --log-ip-options and --log-tcp-options command-line arguments. This allows the resulting iptables syslog messages to include the IP and TCP options portions of the IP and TCP headers if the packet that matches the LOG rule contains them. This functionality is important for both attack detection and passive OS fingerprinting operations performed by psad (see Chapter 7).

The FORWARD Chain

So far the rules we have added to the iptables filtering policy strictly govern the ability of packets to interact directly with the firewall system. Such packets are either destined for or emanate from the firewall operating system and include packets such as connection requests to the SSH daemon from internal systems or locally initiated connections to external sites to download security patches.

Now let's look at the iptables rules that pertain to packets that do not have a source or destination address associated with the firewall, but which nevertheless attempt to route through the firewall system. The iptables `FORWARD` chain in the `filter` table provides the ability to wrap access controls around packets that are forwarded across the firewall interfaces:

```
###### FORWARD chain ######
echo "[+] Setting up FORWARD chain..."
### state tracking rules
$IPTABLES -A FORWARD -m state --state INVALID -j LOG --log-prefix "DROP
INVALID " --log-ip-options --log-tcp-options
$IPTABLES -A FORWARD -m state --state INVALID -j DROP
$IPTABLES -A FORWARD -m state --state ESTABLISHED,RELATED -j ACCEPT

### anti-spoofing rules
$IPTABLES -A FORWARD -i eth1 -s ! $INT_NET -j LOG --log-prefix "SPOOFED PKT "
$IPTABLES -A FORWARD -i eth1 -s ! $INT_NET -j DROP

### ACCEPT rules
❽ $IPTABLES -A FORWARD -p tcp -i eth1 -s $INT_NET --dport 21 --syn -m state
--state NEW -j ACCEPT
$IPTABLES -A FORWARD -p tcp -i eth1 -s $INT_NET --dport 22 --syn -m state
--state NEW -j ACCEPT
$IPTABLES -A FORWARD -p tcp -i eth1 -s $INT_NET --dport 25 --syn -m state
--state NEW -j ACCEPT
$IPTABLES -A FORWARD -p tcp -i eth1 -s $INT_NET --dport 43 --syn -m state
--state NEW -j ACCEPT
$IPTABLES -A FORWARD -p tcp --dport 80 --syn -m state --state NEW -j ACCEPT
$IPTABLES -A FORWARD -p tcp --dport 443 --syn -m state --state NEW -j ACCEPT
$IPTABLES -A FORWARD -p tcp -i eth1 -s $INT_NET --dport 4321 --syn -m state
--state NEW -j ACCEPT
$IPTABLES -A FORWARD -p udp --dport 53 -m state --state NEW -j ACCEPT
$IPTABLES -A FORWARD -p icmp --icmp-type echo-request -j ACCEPT

### default log rule
$IPTABLES -A FORWARD -i ! lo -j LOG --log-prefix "DROP " --log-ip-options
--log-tcp-options
```

Similar to the rules of the `OUTPUT` chain, at ❽ FTP, SSH, SMTP, and whois connections are allowed to be initiated out through the firewall, except that such connections must originate from the internal subnet on the subnet-facing interface (eth1). HTTP, HTTPS, and DNS traffic is allowed through

from any source because we need to allow external addresses to interact with the internal web- and DNS servers (after being NATed; see the following section, "Network Address Translation").

Network Address Translation

The final step in the construction of our iptables policy is to enable the translation of the non-routable 192.168.10.0/24 internal addresses into the routable external 71.157.*X.X* address. This applies to inbound connections to the web- and DNS servers from external clients, and also to outbound connections initiated from the systems on the internal network. For connections initiated from internal systems, we'll use the source NAT (`SNAT`) target, and for connections that are initiated from external systems, we'll use the destination NAT (`DNAT`) target.

The iptables `nat` table is dedicated to all NAT rules, and within this table there are two chains: `PREROUTING` and `POSTROUTING`. The `PREROUTING` chain is used to apply rules in the `nat` table to packets that have not yet gone through the routing algorithm in the kernel in order to determine the interface on which they should be transmitted. Packets that are processed in this chain have also not yet been compared against the `INPUT` or `FORWARD` chains in the `filter` table.

The `POSTROUTING` chain is responsible for processing packets once they have gone through the routing algorithm in the kernel and are just about to be transmitted on the calculated physical interface. Packets processed by this chain have passed the requirements of the `OUTPUT` or `FORWARD` chains in the `filter` table (as well as requirements mandated by other tables that may be registered, such as the `mangle` table).

NOTE *For a complete explanation of how iptables does NAT, see http://www.netfilter.org/documentation/HOWTO/NAT-HOWTO.html.*

```
###### NAT rules ######
echo "[+] Setting up NAT rules..."
❾ $IPTABLES -t nat -A PREROUTING -p tcp --dport 80 -i eth0 -j DNAT
--to 192.168.10.3:80
$IPTABLES -t nat -A PREROUTING -p tcp --dport 443 -i eth0 -j DNAT
--to 192.168.10.3:443
$IPTABLES -t nat -A PREROUTING -p tcp --dport 53 -i eth0 -j DNAT
--to 192.168.10.4:53
❿ $IPTABLES -t nat -A POSTROUTING -s $INT_NET -o eth0 -j MASQUERADE
```

Referring to the network diagram in Figure 1-2, the IP addresses of the web- and DNS servers are 192.168.10.3 and 192.168.10.4 in the internal network. The iptables commands required to provide NAT functionality are displayed above (note the restriction of the commands to the `nat` table through the use of the `-t` option). The three `PREROUTING` rules at ❾ allow web services and DNS requests from the external network to be sent to the appropriate internal servers. The final `POSTROUTING` rule at ❿ allows connections that originate from the internal non-routable network and destined for the external Internet to look as though they come from the IP address 71.157.*X.X*.

The very last step in building the iptables policy is to enable IP forwarding in the Linux kernel:

```
###### forwarding ######
echo "[+] Enabling IP forwarding..."
echo 1 > /proc/sys/net/ipv4/ip_forward
```

Activating the Policy

One of the really nice things about iptables is that instantiating a policy within the kernel is trivially easy through the execution of iptables commands—there are no heavyweight user interfaces, binary file formats, or bloated management protocols (like the ones developed by some proprietary vendors of other security products). Now that we have a shell script that captures the iptables commands (once again, you can download the complete script from http://www.cipherdyne.org/LinuxFirewalls), let's execute it:

```
[iptablesfw]# ./iptables.sh
[+] Flushing existing iptables rules...
[+] Setting up INPUT chain...
[+] Setting up OUTPUT chain...
[+] Setting up FORWARD chain...
[+] Setting up NAT rules...
[+] Enabling IP forwarding...
```

iptables-save and iptables-restore

All of the previous iptables commands in the iptables.sh script are executed one at a time in order to instantiate new rules, set the default policy on a chain, or delete old rules. Each command requires a separate execution of the iptables userland binary to create the iptables policy. Hence, this is not an optimal solution for bringing the policy into existence quickly at system boot, particularly when the number of iptables rules grows into the hundreds (which can happen with a policy built by fwsnort, as we will see in Chapter 10). A much faster mechanism is provided by the commands `iptables-save` and `iptables-restore`, which are installed within the same directory (/sbin in our case) as the main iptables program. The `iptables-save` command builds a file that contains all iptables rules in a running policy in human-readable format. This format can be interpreted by the `iptables-restore` program, which takes each of the rules listed in the ipt.save file and instantiates it within a running kernel. A single execution of the `iptables-restore` program recreates an entire iptables policy in the kernel; multiple executions of the iptables program are not necessary. This makes the `iptables-save` and `iptables-restore` commands ideal for rapid deployment of iptables rulesets, and I illustrate this process with the following two commands:

```
[iptablesfw]# iptables-save > /root/ipt.save
[iptablewfw]# cat /root/ipt.save | iptables-restore
```

The contents of the ipt.save file are organized by iptables table, and within each section devoted to an individual table, ipt.save is further organized by iptables chain. A line that begins with an asterisk (*) character followed by a table name (such as `filter`) denotes the beginning of a section in the ipt.save file that describes a particular table. Following this are lines that track packet and bytes counts for each chain associated with the table.

The next portion of the ipt.save file is a complete description of all iptables rules organized by chain. These lines allow the actual iptables ruleset to be reconstructed by `iptables-restore`; even including packet and byte counts for each rule if the `-c` option to `iptables-save` is used.

Lastly, the word `COMMIT` on a line by itself concludes the section of the ipt.save file that characterizes the iptables table. This line constitutes the ending marker for all information associated with the table. Below is a complete example of what the `filter` table section looks like once we have executed all of the iptables commands up to this point in the chapter:

```
# Generated by iptables-save v1.3.7 on Sat Apr 14 17:35:22 2007
*filter
:INPUT DROP [0:0]
:FORWARD DROP [0:0]
:OUTPUT DROP [2:112]
-A INPUT -m state --state INVALID -j LOG --log-prefix "DROP INVALID "
--log-tcp-options --log-ip-options
-A INPUT -m state --state INVALID -j DROP
-A INPUT -m state --state RELATED,ESTABLISHED -j ACCEPT
-A INPUT -s ! 192.168.10.0/255.255.255.0 -i eth1 -j LOG --log-prefix
"SPOOFED PKT "
-A INPUT -s ! 192.168.10.0/255.255.255.0 -i eth1 -j DROP
-A INPUT -s 192.168.10.0/255.255.255.0 -i eth1 -p tcp -m tcp --dport 22
--tcp-flags FIN,SYN,RST,ACK SYN -m state --state NEW -j ACCEPT
-A INPUT -p icmp -m icmp --icmp-type 8 -j ACCEPT
-A INPUT -i ! lo -j LOG --log-prefix "DROP " --log-tcp-options
--log-ip-options
-A FORWARD -m state --state INVALID -j LOG --log-prefix "DROP INVALID "
--log-tcp-options --log-ip-options
-A FORWARD -m state --state INVALID -j DROP
-A FORWARD -m state --state RELATED,ESTABLISHED -j ACCEPT
-A FORWARD -s ! 192.168.10.0/255.255.255.0 -i eth1 -j LOG
--log-prefix "SPOOFED PKT "
-A FORWARD -s ! 192.168.10.0/255.255.255.0 -i eth1 -j DROP
-A FORWARD -s 192.168.10.0/255.255.255.0 -i eth1 -p tcp -m tcp --dport 21
--tcp-flags FIN,SYN,RST,ACK SYN -m state --state NEW -j ACCEPT
-A FORWARD -s 192.168.10.0/255.255.255.0 -i eth1 -p tcp -m tcp --dport 22
--tcp-flags FIN,SYN,RST,ACK SYN -m state --state NEW -j ACCEPT
-A FORWARD -s 192.168.10.0/255.255.255.0 -i eth1 -p tcp -m tcp --dport 25
--tcp-flags FIN,SYN,RST,ACK SYN -m state --state NEW -j ACCEPT
-A FORWARD -p tcp -m tcp --dport 80 --tcp-flags FIN,SYN,RST,ACK SYN -m state
--state NEW -j ACCEPT
-A FORWARD -p tcp -m tcp --dport 443 --tcp-flags FIN,SYN,RST,ACK SYN -m state
--state NEW -j ACCEPT
-A FORWARD -p udp -m udp --dport 53 -m state --state NEW -j ACCEPT
-A FORWARD -p icmp -m icmp --icmp-type 8 -j ACCEPT
```

```
-A FORWARD -i ! lo -j LOG --log-prefix "DROP " --log-tcp-options
--log-ip-options
-A OUTPUT -m state --state INVALID -j LOG --log-prefix "DROP INVALID "
--log-tcp-options --log-ip-options
-A OUTPUT -m state --state INVALID -j DROP
-A OUTPUT -m state --state RELATED,ESTABLISHED -j ACCEPT
-A OUTPUT -p tcp -m tcp --dport 21 --tcp-flags FIN,SYN,RST,ACK SYN -m state
--state NEW -j ACCEPT
-A OUTPUT -p tcp -m tcp --dport 22 --tcp-flags FIN,SYN,RST,ACK SYN -m state
--state NEW -j ACCEPT
-A OUTPUT -p tcp -m tcp --dport 25 --tcp-flags FIN,SYN,RST,ACK SYN -m state
--state NEW -j ACCEPT
-A OUTPUT -p tcp -m tcp --dport 43 --tcp-flags FIN,SYN,RST,ACK SYN -m state
--state NEW -j ACCEPT
-A OUTPUT -p tcp -m tcp --dport 80 --tcp-flags FIN,SYN,RST,ACK SYN -m state
--state NEW -j ACCEPT
-A OUTPUT -p tcp -m tcp --dport 443 --tcp-flags FIN,SYN,RST,ACK SYN -m state
--state NEW -j ACCEPT
-A OUTPUT -p tcp -m tcp --dport 4321 --tcp-flags FIN,SYN,RST,ACK SYN -m state
--state NEW -j ACCEPT
-A OUTPUT -p udp -m udp --dport 53 -m state --state NEW -j ACCEPT
-A OUTPUT -p icmp -m icmp --icmp-type 8 -j ACCEPT
-A OUTPUT -o ! lo -j LOG --log-prefix "DROP " --log-tcp-options
--log-ip-options
COMMIT
# Completed on Sat Apr 14 17:35:22 2007
```

At this point we have a functional iptables policy that maintains a high level of control over the packets that attempt to traverse the firewall interfaces, and we have a convenient way to rapidly reinstantiate this policy by executing the `iptables-restore` command against the ipt.save file. This has obvious applications for accelerating the system boot cycle, but it is also useful for testing new policies, since it makes it extremely easy to revert to a known-good state. There is one thing missing, however: Altering the iptables policy is most easily accomplished by editing a script instead of by editing the ipt.save file directly (which has a strict syntax requirement that is not as widely known as, say, a Bourne shell script).

Testing the Policy: TCP

Once an iptables policy has been created within the Linux kernel and basic connectivity through the firewall has been verified, it is a good idea to test the policy in order to make sure there are no chinks in the virtual armor. It is most important to test the iptables policy from a host that is external to the local network, because this is the source of the majority of attacks (assuming a huge number of users are not on the internal systems). Effective testing is also important from the internal network, however, since one of the internal hosts could be compromised and then used to attack other internal hosts (including the firewall), even though iptables is protecting the entire network.

Client-side vulnerabilities, such as the Microsoft JPEG vulnerability,[6] make this a realistic possibility if there are unpatched systems on the internal network.

To begin testing the policy, we first test access to TCP ports that should not be accessible from the either the internal or external networks. Recall that RFC 793 requires a properly implemented TCP stack to generate a reset (RST/ACK[7]) packet if a SYN packet is received on closed port. This provides us with an easy way to verify that iptables is actually blocking packets, since the absence of a RST/ACK packet in response to a connection attempt would indicate that iptables has intercepted the SYN packet within the kernel and has not allowed the TCP stack to generate the RST/ACK back to the client. We randomly select TCP port 5500 to test from both internal and external hosts. The following example illustrates this test and shows that the iptables `INPUT` chain is indeed functioning correctly, since not only are the packets dropped, but the appropriate log messages are also generated. First we test from the ext_scanner system by using Netcat to attempt to connect to TCP port 5500 on the firewall. As expected, the Netcat client just hangs, and on the firewall itself, a log message is generated indicating that iptables intercepted and dropped a TCP SYN packet to port 5500:

```
[ext_scanner]$ nc -v 71.157.X.X 5500
[iptablesfw]# tail /var/log/messages |grep 5500
Apr 14 16:52:43 iptablesfw kernel: DROP IN=eth0 OUT=
MAC=00:13:d3:38:b6:e4:00:30:48:80:4e:37:08:00 SRC=144.202.X.X DST=71.157.X.X
LEN=60 TOS=0x00 PREC=0x00 TTL=64 ID=54983 DF PROTO=TCP SPT=59604 DPT=5500
WINDOW=5840 RES=0x00 SYN URGP=0 OPT (020405B40402080A1E92414600000000001030306)
```

NOTE *The above iptables log message is the first in the book, and you may have trouble making sense of it. I will cover iptables log messages in detail (and with an eye toward recognizing suspicious traffic) in Chapters 2 and 3.*

Similarly, we get the same results from the internal network:

```
[int_scanner]$ nc -v 192.168.10.1 5500
[iptablesfw]# tail /var/log/messages |grep 5500 |tail -n 1
Apr 14 16:55:53 iptablesfw kernel: DROP IN=eth1 OUT=
MAC=00:13:46:3a:41:4b:00:a0:cc:28:42:5a:08:00 SRC=192.168.10.200
DST=192.168.10.1 LEN=60 TOS=0x10 PREC=0x00 TTL=64 ID=4858 DF PROTO=TCP
SPT=58715 DPT=5500 WINDOW=5840 RES=0x00 SYN URGP=0 OPT
(020405B40402080A0039F4D30000000001030305)
```

If we had received a RST/ACK packet in either of the tests in the above code example (which would indicate that iptables had not intercepted the SYN packet before it had a chance to interact with the TCP stack running on the firewall), Netcat would have displayed the message `Connection refused`.

[6] See http://www.securityfocus.com/archive/1/375204/2004-09-09/2004-09-15/0 for more information.

[7] The details regarding whether or not a RST packet has the ACK bit set are discussed in detail in Chapter 3.

NOTE *It's a good idea to run Nmap against the firewall to rigorously test the iptables policy. Nmap offers many different scanning types that assist in making sure that the connection-tracking and filtering capabilities offered by iptables are doing their jobs. For example, sending a surprise FIN packet (see Nmap's* `-sF` *scanning mode) against a closed port should not elicit a RST/ACK packet if iptables is working properly. Generating TCP ACK packets that are not part of any established session (Nmap's* `-sA` *mode) should similarly be met with utter silence, because the connection-tracking subsystem is able to discern that such packets are not part of any legitimate TCP session.*

Testing the Policy: UDP

Next, we'll test iptables's ability to filter against UDP ports. Servers that run over UDP sockets exist in a different world than those that run over TCP sockets. UDP is a connectionless protocol, and so there is no notion analogous to a TCP handshake or even a scheme to acknowledge data in UDP traffic. Similar constructs such as reliable data delivery can be built in to applications that run over UDP, but this requires application-level modifications, whereas TCP has these features built in for free. UDP simply throws packets out on the network and hopes they reach the intended destination.

To show that iptables is indeed working properly for UDP traffic, we send packets to UDP port 5500 again from both internal and external systems, just as we did for TCP. However, this time, if our UDP packet is not filtered, we should receive an ICMP Port Unreachable message back to our client. This time, we use the hping utility (see http://www.hping.org). In both cases of the external and internal hosts trying to talk to the UDP stack running on the firewall, iptables correctly intercepts the packets. First we test from the external host:

```
[ext_scanner]# hping -2 -p 5500 71.157.X.X
HPING 71.157.X.X (eth0 71.157.X.X): udp mode set, 28 headers + 0 data bytes
[iptablesfw]# tail /var/log/messages |grep 5500
Apr 14 16:58:31 iptablesfw kernel: DROP IN=eth0 OUT=
MAC=00:13:d3:38:b6:e4:00:30:48:80:4e:37:08:00 SRC=144.202.X.X DST=71.157.X.X
LEN=28 TOS=0x00 PREC=0x00 TTL=64 ID=22084 PROTO=UDP SPT=2202 DPT=5500 LEN=8
```

Similarly, we achieve the same result for the internal network:

```
[int_scanner]# hping -2 -p 5500 192.168.10.1
HPING 192.168.10.1 (eth0 192.168.10.1): udp mode set, 28 headers + 0 data
bytes
[iptablesfw]# tail /var/log/messages |grep 5500 |tail -n 1
Apr 14 17:00:24 iptablesfw kernel: DROP IN=eth1 OUT=
MAC=00:13:46:3a:41:4b:00:a0:cc:28:42:5a:08:00 SRC=192.168.10.200
DST=192.168.10.1 LEN=28 TOS=0x00 PREC=0x00 TTL=64 ID=35261 PROTO=UDP SPT=2647
DPT=5500 LEN=8
```

NOTE *This brings up an interesting observation about security: In these tests, any unprivileged user could have used Netcat to listen on TCP or UDP port 5500, but we would have been completely unable to access the server from any IP address that is not explicitly allowed through by the iptables policy. This means that any server started on the system cannot adversely affect the overall security of the system (at least from remote attacks) without also modifying the iptables policy. This is a powerful concept that helps to make the case that a firewall should be deployed on every system; the additional work that is created by having to manage the firewall policy is well worth the effort in the face of risking potential compromise.*

Testing the Policy: ICMP

Finally, we'll test the iptables policy over ICMP. The iptables commands used in the construction of the policy used the `--icmp-type` option to restrict acceptable ICMP packets to just Echo Request packets (the connection-tracking code allows the corresponding Echo Reply packets to be sent so an explicit `ACCEPT` rule does not have to be added to allow such replies). Therefore, iptables should be allowing all Echo Request packets, but other ICMP packets should be met with stark silence. We test this by generating ICMP Echo Reply packets without sending any corresponding Echo Request packets, which should cause iptables to match the packets on the `INVALID` state rule at the beginning of the `INPUT` chain. Again, we turn to hping to test from both the internal and external networks. The first test is to generate an unsolicited ICMP Echo Reply packet from the external network, and we expect that iptables will log and drop the packet in the `INPUT` chain. By examining the iptables log, we see that this is indeed the case (the `DROP INVALID` log prefix is in bold):

```
[ext_scanner]# hping -1 --icmptype echo-reply 71.157.X.X
HPING (eth1 71.157.X.X): icmp mode set, 28 headers + 0 data bytes
--- 71.157.X.X hping statistic ---
2 packets transmitted, 0 packets received, 100% packet loss
round-trip min/avg/max = 0.0/0.0/0.0 ms
[iptablesfw]# tail /var/log/messages |grep ICMP
Apr 14 17:04:58 iptablesfw kernel: DROP INVALID IN=eth0 OUT=
MAC=00:13:d3:38:b6:e4:00:30:48:80:4e:37:08:00 SRC=144.202.X.X DST=71.157.X.X
LEN=28 TOS=0x00 PREC=0x00 TTL=64 ID=44271 PROTO=ICMP TYPE=0 CODE=0 ID=21551
SEQ=0
```

Similarly, the same result is achieved from the internal network:

```
[int_scanner]# hping -1 --icmptype echo-reply 192.168.10.1
HPING (eth1 192.168.10.1): icmp mode set, 28 headers + 0 data bytes
--- 192.168.10.1 hping statistic ---
2 packets transmitted, 0 packets received, 100% packet loss
round-trip min/avg/max = 0.0/0.0/0.0 ms
[iptablesfw]# tail /var/log/messages |grep ICMP |tail -n 1
Apr 14 17:06:45 iptablesfw kernel: DROP INVALID IN=eth1 OUT=
MAC=00:13:46:3a:41:4b:00:a0:cc:28:42:5a:08:00 SRC=192.168.10.200
DST=192.168.10.1 LEN=28 TOS=0x00 PREC=0x00 TTL=64 ID=36520 PROTO=ICMP TYPE=0
CODE=0 ID=44313 SEQ=0
```

Concluding Thoughts

This chapter focuses on iptables concepts that are important for the rest of the book and lays a foundation from which to begin discussing intrusion detection and response from an iptables standpoint. We are now armed with a default iptables policy and network diagram that is referenced in several upcoming chapters, and we have seen examples of iptables log messages that illustrate the completeness of the iptables logging format. We are now ready to jump into a treatment of attacks that we can detect—and thwart, as we shall see—with iptables.