# 3

# Building and Installing a Firewall

CHAPTER 2, "PACKET-FILTERING CONCEPTS," COVERS the background ideas and concepts behind a packet-filtering firewall. Each firewall rule chain has its own default policy. Each rule applies not only to an individual `input` or `output` chain, but to a specific network interface, message protocol type (i.e., TCP, UDP, ICMP), and service port number. Individual acceptance, denial, and rejection rules are defined for the `input` chain and the `output` chain, as well as for the `forward` chain, which you'll learn about at the end of this chapter and in Chapter 4, "LAN Issues, Multiple Firewalls, and Perimeter Networks." This chapter pulls those ideas together to demonstrate how to build a simple, single-system, custom-designed firewall for your site.

The firewall you'll build in this chapter is based on a deny-everything-by-default policy. All network traffic is blocked by default. Services are individually enabled as exceptions to the policy.

After the single-system firewall is built, the chapter closes by demonstrating how to extend the standalone firewall to a formal bastion firewall. A *bastion firewall* has at least two network interfaces. It insulates an internal LAN from direct communication with the Internet. Only minor extensions are required to make the single-system firewall function as a simple, dual-homed bastion firewall. It protects your internal LAN by applying packet-filtering rules at the external interface, acting as a proxying gateway between the LAN and the Internet.

The single-system and bastion firewalls are the least-secure forms of firewall architecture. If the firewall host were to be compromised, any local machines would be open to attack. As a standalone firewall, it's an all-or-nothing proposition. Because this book is targeted at the home and small business user, the assumption is that the majority of home users have a single computer connected to the Internet, or a single firewall machine protecting a small, private LAN. "Least-secure," however, does not imply an insecure firewall. These firewalls are less flexible than more complicated architectures involving multiple machines. Chapter 4 introduces more flexible configurations that allow for additional internal security protecting more complicated LAN and server configurations than a single-system firewall can.

## *ipchains*: The Linux Firewall Administration Program

This book is based on Red Hat Linux 6.0. Linux comes supplied with a firewall mechanism called IPFW (IP firewall). The major Linux distributions either have or are in the process of converting to `ipchains`, a rewrite of IPFW version 4. This new version is usually referred to as `ipchains`, its administration program's name. Previous Linux distributions used an earlier IPFW implementation. Their firewall mechanism is usually referred to as `ipfwadm`, the earlier version's administration program's name.

`ipchains` is used in the examples in this book. Because `ipfwadm` remains in widespread use on older Linux systems, `ipfwadm` versions of the examples are presented in Appendix B, "Firewall Examples and Support Scripts." Although the syntax differs between `ipchains` and `ipfwadm`, they are functionally the same. `ipchains` includes the `ipfwadm` feature set, along with additional features found in the new IPFW implementation. The features new to `ipchains` won't be used in this book. Only the features common to both versions are used in the examples.

As a firewall administration program, `ipchains` creates the individual packet filter rules for the `input` and `output` chains composing the firewall. One of the most important aspects of defining firewall rules is that the order in which the rules are defined is important.

Packet-filtering rules are stored in kernel tables, in an `input`, `output`, or `forward` chain, in the order in which they are defined. Individual rules are inserted at the beginning of the chain or appended to the end of the chain. All rules are appended in the examples in this chapter (with one exception at the end of the chapter). The order you define the rules in is the order they'll be added to the kernel tables, and thereby the order the rules will be compared against each packet.

As each externally originating packet arrives at a network interface, its header fields are compared against each rule in the interface's `input` chain until a match is found. Conversely, as each internally originating packet is sent to a network interface, its header fields are compared against each rule in the interface's `output` chain until a match is found. In either direction, when a match is found, the comparison stops and

the rule's packet disposition is applied: `ACCEPT`, `REJECT`, or `DENY`. If the packet doesn't match any rule on the chain, the default policy for that chain is applied. The bottom line is that *the first matching rule wins*.

The numeric service port numbers, rather than their symbolic names, as listed in `/etc/services`, are used in all the examples of rules in this chapter. `ipchains` supports the symbolic service port names. The examples in this chapter use the numeric values because the symbolic names are not consistent across Linux distributions, or even from one release to the next. You could use the symbolic names for clarity in your own rules, but remember that your firewall could break with the next system upgrade.

`ipchains` is a compiled C program. It must be invoked once for each individual firewall rule you define. This is done from a shell script. The examples in this chapter assume that the shell script is named `/etc/rc.d/rc.firewall`. In cases where shell semantics differ, the examples are written in Bourne (`sh`), Korn (`ksh`), or Bourne Again (`bash`) shell semantics.

The examples are not optimized. They are spelled out for clarity and to maintain conceptual and feature-set compatibility between `ipchains` and `ipfwadm`. The two programs use different command-line arguments to reference similar features, and offer slightly different shortcuts and optimization capabilities. The examples are presented using the features common to both programs.

## *ipchains* Options Used in the Firewall Script

`ipchains` options aren't covered in full in this chapter. Only the features used in the examples in this book, features common to `ipfwadm`, are covered. Table 3.1 lists the `ipchains` command-line arguments used here:

```
ipchains -A¦I [chain] [-i interface] [-p protocol] [ [!] -y]
         [-s address [port[:port]]]
         [-d address [port[:port]]]
         -j policy [-l]
```

> For a description of the complete `ipchains` feature set, refer to the online `man` page, `ipchains`, and to `IPCHAINS-HOWTO`.

**Table 3.1 *ipchains* Options Used in the Firewall Script**

| Option | Description |
|---|---|
| `-A [chain]` | Append a rule to the end of a chain. The examples use the built-in chains: `input`, `output`, and `forward`. If a chain isn't specified, the rule applies to all chains. |
| `-I [chain]` | Insert a rule at the beginning of a chain. The examples use the built-in chains: `input`, `output`, and `forward`. If a chain isn't specified, the rule applies to all chains. |
| `-i <interface>` | Specify the network interface the rule applies to. If an interface isn't specified, the rule applies to all interfaces. Common interface names are `eth0`, `eth1`, `lo`, and `ppp0`. |
| `-p <protocol>` | Specify the IP protocol the rule applies to. If the `-p` option isn't used, the rule applies to all protocols. The supported protocol names are `tcp`, `udp`, `icmp`, and `all`. Any of the protocol names or numbers from `/etc/protocols` are also allowed. |
| `-y` | The `SYN` flag must be set, and the `ACK` flag must be cleared in a TCP message, indicating a connection establishment request. If `-y` isn't given as an argument, the TCP flag bits aren't checked. |
| `! -y` | The `ACK` flag must be set in a TCP message, indicating an initial response to a connection request or an ongoing, established connection. If `! -y` isn't given as an argument, the TCP flag bits aren't checked. |
| `-s <address> [<port>]` | Specify the packet's source address. If a source address isn't specified, all unicast source addresses are implied. If a port or range of ports is given, the rule applies to only those ports. Without a port specifier, the rule applies to all source ports. A range of ports is defined by a beginning and ending port number, separated by a colon (e.g., `1024:65535`). If a port is given, an address must be specified. |
| `-d <address> [<port>]` | Specify the packet's destination address. If a destination address isn't specified, all unicast destination addresses are implied. If a port or range of ports is given, the rule applies to only those ports. Without a port specifier, the rule applies to all destination ports. A range of ports is defined by a beginning and ending port number, separated by a colon (e.g., `1024:65535`). If a port is given, an address must be specified. |

| Option | Description |
|---|---|
| `-j <policy>` | Specify the packet disposition policy for this rule: `ACCEPT`, `DENY`, or `REJECT`. The `forward` chain can take the `MASQ` (masquerade) policy, as well. |
| `-l` | Write a kernel informational (`KERN_INFO`) message in the system log, `/var/log/messages` by default, whenever a packet matches this rule. |

## Source and Destination Addressing Options

Both a packet's source and destination addresses can be specified in a firewall rule. Only packets with that specific source or destination address match the rule. Addresses may be a specific IP address, a fully qualified hostname, a network (domain) name, a limited range of addresses, or all-inclusive.

An IP address is a 32-bit numeric value, divided into four individual 8-bit bytes, ranging from 0 through 255. In dotted-decimal notation, each of the four bytes making up the 32-bit value is represented as one of the quads in the IP address. The private Class C IP address `192.168.10.30` is used as the local host address in the figures throughout this book.

`ipchains` allows the address to be suffixed with a bit mask specifier. The mask's value can range from 0 through 32, indicating the number of bits to mask. Bits are counted from the left, or most significant, quad. This mask specifier indicates how many of the leading bits in the address must exactly match the IP address defined in the rule.

A mask of 32, `/32`, means that all the bits must match. The address must exactly match what you've defined in the rule. Specifying an address as `192.168.10.30` is the same as specifying the address as `192.168.10.30/32`. The `/32` mask is implied by default. You don't need to specify it.

### IP Addresses Expressed As Symbolic Names

Remote hosts and networks may be specified as fully qualified host or network names. Using a hostname is especially convenient for firewall rules that apply to an individual remote host. This is particularly true for hosts whose IP address can change, or that invisibly represent multiple IP addresses, such as ISP mail servers sometimes do. In general, however, remote addresses are better expressed in dotted quad notation because of the possibility of DNS hostname spoofing.

Symbolic hostnames can't be resolved until DNS traffic is enabled in firewall rules. If hostnames are used in the firewall rules, those rules must follow the rules enabling DNS traffic.

An example using masking would be to allow only a given connection type to be made between you and your ISP's server machines. Let's say that your ISP uses addresses in the range of `192.168.24.0` through `192.168.27.255` for its server address space. In this case, the address/mask pair would be `192.168.24/22`. As shown in Figure 3.1, the first 22 bits of all addresses in this range are identical, so any address matching on the first 22 bits will match. Effectively, you are saying that you will allow connections to the service only when offered from machines in the address range `192.168.24.0` through `192.168.27.255`.

A mask of 0, `/0`, means that no bits in the address are required to match. In other words, because no bits need to match, using `/0` is the same as not specifying an address. Any unicast address matches. `ipchains` has a built-in alias for `0.0.0.0/0`, `any/0`.

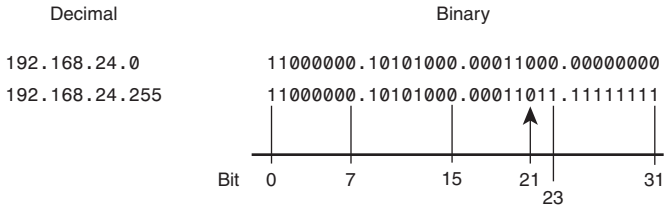


**Figure 3.1** The first matching 22 bits in the masked IP address range `192.168.24.0/22`.

# Initializing the Firewall

A firewall is implemented as a series of packet-filtering rules defined by options on the `ipchains` command line. `ipchains` is executed once for each individual rule. (Different firewalls can range from a dozen rules to hundreds.)

The `ipchains` invocations should be made from an executable shell script and not directly from the command line. You should invoke the complete firewall shell script. Do not attempt to invoke specific `ipchains rules` from the command line because this could cause your firewall to accept or deny packets inappropriately. When the chains are initialized and the default deny policy is enabled, all network services are blocked until an acceptance filter is defined to allow the individual service.

Likewise, you should execute the shell script from the console. Do not execute the shell script from either a remote machine or from an X Window `xterm` session. Not only is remote network traffic blocked, but access to the local loopback interface used by X Window is blocked until access to the interface is explicitly reenabled.

Furthermore, remember that firewall filters are applied in the order in which you've defined them on the `input` or `output` chain. The rules are appended to the end of their chain in the order you define them. The first matching rule wins. Because of this, firewall rules must be defined in a hierarchical order from most specific to more general rules.

*Firewall initialization* is used to cover a lot of ground, including defining global constants used in the shell script, clearing out any existing rules in the firewall chains, defining default policies for the `input` and `output` chains, reenabling the loopback interface for normal system operation, denying access from any specific hosts or networks you've decided to block, and defining some basic rules to protect against bad addresses and to protect certain services running on unprivileged ports.

## Symbolic Constants Used in the Firewall Examples

A firewall shell script is easiest to read and maintain if symbolic constants are used for recurring names and addresses. The following constants are either used throughout the examples in this chapter, or else are universal constants defined in the networking standards:

```
EXTERNAL_INTERFACE="eth0"            # Internet-connected interface
LOOPBACK_INTERFACE="lo"              # however your system names it

IPADDR="my.ip.address"               # your IP address
ANYWHERE="any/0"                     # match any IP address
MY_ISP="my.isp.address.range"        # ISP server & NOC address range

LOOPBACK="127.0.0.0/8"               # reserved loopback address range
CLASS_A="10.0.0.0/8"                 # class A private networks
CLASS_B="172.16.0.0/12"              # class B private networks
CLASS_C="192.168.0.0/16"             # class C private networks
CLASS_D_MULTICAST="224.0.0.0/4"      # class D multicast addresses
CLASS_E_RESERVED_NET="240.0.0.0/5"   # class E reserved addresses
```

```
BROADCAST_SRC="0.0.0.0"              # broadcast source address
BROADCAST_DEST="255.255.255.255"     # broadcast destination address
PRIVPORTS="0:1023"                   # well-known, privileged port range
UNPRIVPORTS="1024:65535"             # unprivileged port range
```

Constants not listed here are defined within the context of the specific rules they are used with.

## Removing Any Preexisting Rules

The first thing to do when defining a set of filtering rules is to remove any existing rules from their chain. Otherwise, any new rules you define would be added to the end of existing rules. Packets could easily match a preexisting rule before reaching the point in the chain you are defining from this point on.

Removal is called *flushing* the chain. Without a directional argument referring to a specific chain, the following command flushes the rules of all three built-in chains—`input`, `output`, and `forward`—at once:

```
# Flush any existing rules from all chains
ipchains -F
```

The chains are empty. You're starting from scratch. The system is in its default accept-everything policy state.

## Defining the Default Policy

A side effect of flushing all the rules is that the system is returned to its default state, including the default accept-everything policy for each chain. Until new default policies are defined, the system allows everything through the network interfaces. No filtering is done.

By default, you want the firewall to deny everything coming in and reject everything going out. Unless a rule is defined to explicitly allow a matching packet through, incoming packets are silently denied without notification to the remote sender, and outgoing packets are rejected and an ICMP error message is returned to the local sender. The difference for the end user is that, for example, if someone at a remote site attempts to connect to your Web server, that person's browser hangs until his or her system returns a TCP timeout condition. He or she has no indication whether your site or your Web server exist. If you, on the other hand, attempt to connect to a remote Web server, your browser receives an immediate error condition indicating that the operation isn't allowed:

```
# Set the default policy to deny
ipchains -P input   DENY
ipchains -P output  REJECT
ipchains -P forward REJECT
```

At this point, all network traffic is blocked.

## Enabling the Loopback Interface

You need to enable unrestricted loopback traffic. This allows you to run any local network services you choose—or that the system depends on—without having to worry about getting all the firewall rules specified.

Loopback is enabled immediately in the firewall script. It's not an externally available interface. Local network-based services, such as the X Window system, will hang until loopback traffic is allowed through.

The rules are simple when everything is allowed. You simply need to undo the effect of the default deny policies for the loopback interface by accepting everything on that interface:

```
# Unlimited traffic on the loopback interface
ipchains -A input  -i $LOOPBACK_INTERFACE -j ACCEPT
ipchains -A output -i $LOOPBACK_INTERFACE -j ACCEPT
```

System logging, X Window, and other local, UNIX-domain, socket-based services are available again.

## Source Address Spoofing and Other Bad Addresses

This section establishes some `input` chain filters based on source and destination addresses. These addresses will never be seen in a legitimate incoming packet from the Internet.

The Linux kernel offers some support against incoming spoofed packets in addition to what can be done at the firewall level. Also, in case TCP `SYN` Cookie protection is not enabled, the following lines enable both kernel support modules:

```
echo 1 >/proc/sys/net/ipv4/tcp_syncookies

# Setting up IP spoofing protection
# turn on Source Address Verification
for f in /proc/sys/net/ipv4/conf/*/rp_filter; do
    echo 1 > $f
done
```

### Default Policy Rules and the First Matching Rule Wins

The default policies appear to be exceptions to the first-matching-rule-wins scenario. The default policy commands are not position-dependent. They aren't rules, per se. A chain's default policy is applied after a packet has been compared to each rule on the chain without a match.

The default policies are defined first in the script to define the default packet disposition before any rules to the contrary are defined. If the policy commands were executed at the end of the script, and the firewall script contained a syntax error causing it to exit prematurely, the default accept-everything policy would be in effect. If a packet didn't match a rule, (and rules are usually accept rules in a deny-everything-by-default firewall) the packet would fall off the end of the chain and be accepted by default. The firewall rules would not be accomplishing anything useful.

At the packet-filtering level, one of the few cases of source address spoofing you can identify with certainty as a forgery is your own IP address. This rule denies incoming packets claiming to be from you:

```
# Refuse spoofed packets pretending to be from
# the external interface's IP address
ipchains -A input  -i $EXTERNAL_INTERFACE -s $IPADDR -j DENY -l
```

There is no need to block outgoing packets destined to yourself. They won't return, claiming to be from you and appearing to be spoofed. Remember, if you send packets to your own external interface, those packets arrive on the loopback interface's input queue, not on the external interface's input queue. Packets containing your address as the source address never arrive on the external interface, even if you send packets to the external interface.

As explained in Chapter 2, spare private IP addresses are set aside in each of the Class A, B, and C address ranges for use in private LANs. They are not intended for use on the Internet. Routers are not supposed to route packets with private source addresses. Routers cannot route packets with private destination addresses. Nevertheless, many routers do allow packets through with private source addresses.

Additionally, if someone on your ISP's subnet (i.e., on your side of the router you share) is leaking packets with private destination IP addresses, you'll see them even if the router doesn't forward them. Machines on your own LAN could also leak private source addresses if your IP masquerading or proxy configuration is set up incorrectly.

The next three sets of rules disallow incoming and outgoing packets with any of the Class A, B, or C private network addresses as their source or destination addresses.

**Firewall Logging**

The `-l` option enables logging for packets matching the rule. When a packet matches the rule, the event is logged in `/var/log/messages`. Firewall logging is available by default in Red Hat 6.0. Releases prior to version 6.0 required you to recompile the kernel with the kernel logging module included in order to use the `ipchains/ipfwadm` logging option.

None of these packets should be seen outside a private LAN:

```
# Refuse packets claiming to be to or from a Class A private network
ipchains -A input  -i $EXTERNAL_INTERFACE -s $CLASS_A -j DENY
ipchains -A input  -i $EXTERNAL_INTERFACE -d $CLASS_A -j DENY
ipchains -A output -i $EXTERNAL_INTERFACE -s $CLASS_A -j DENY -l
ipchains -A output -i $EXTERNAL_INTERFACE -d $CLASS_A -j DENY -l

# Refuse packets claiming to be to or from a Class B private network
ipchains -A input  -i $EXTERNAL_INTERFACE -s $CLASS_B -j DENY
ipchains -A input  -i $EXTERNAL_INTERFACE -d $CLASS_B -j DENY
ipchains -A output -i $EXTERNAL_INTERFACE -s $CLASS_B -j DENY -l
ipchains -A output -i $EXTERNAL_INTERFACE -d $CLASS_B -j DENY -l

# Refuse packets claiming to be to or from a Class C private network
ipchains -A input  -i $EXTERNAL_INTERFACE -s $CLASS_C -j DENY
ipchains -A input  -i $EXTERNAL_INTERFACE -d $CLASS_C -j DENY
ipchains -A output -i $EXTERNAL_INTERFACE -s $CLASS_C -j DENY -l
ipchains -A output -i $EXTERNAL_INTERFACE -d $CLASS_C -j DENY -l
```

Your external network interface won't recognize a destination address other than its own if your routing table is configured correctly. But if you configured automatic routing and had a LAN using these addresses and someone on your ISP's subnet was leaking packets, your firewall could conceivably forward the packets to your LAN.

The next two rules disallow packets with a source address reserved for the loopback interface:

```
# Refuse packets claiming to be from the loopback interface
ipchains -A input  -i $EXTERNAL_INTERFACE -s $LOOPBACK -j DENY-
ipchains -A output -i $EXTERNAL_INTERFACE -s $LOOPBACK -j DENY -l
```

Because loopback addresses are assigned to a local software interface, which system software handles internally, any packet claiming to be from such an address is intentionally forged. Notice that I've chosen to log the event if a local user attempts to spoof the address.

As with addresses set aside for use in private LANs, routers are not supposed to forward packets originating from the loopback address range. A router cannot forward a packet with a loopback destination address.

The next two rules block broadcast packets containing illegal source or destination broadcast addresses. The firewall's default policy is to deny everything. As such, broadcast destination addresses are denied by default and must be explicitly enabled in the cases where they are wanted:

```
# Refuse malformed broadcast packets
ipchains -A input  -i $EXTERNAL_INTERFACE -s $BROADCAST_DEST -j DENY -l
ipchains -A input  -i $EXTERNAL_INTERFACE -d $BROADCAST_SRC  -j DENY -l
```

The first of these rules logs and denies any packet claiming to come from `255.255.255.255`, the address reserved as the broadcast destination address. A packet will never legitimately originate from address `255.255.255.255`.

The second of these rules logs and denies any packet directed to destination address `0.0.0.0`, the address reserved as a broadcast source address. Such a packet is not a mistake; it is a specific probe intended to identify a UNIX machine running network software derived from BSD. Because most UNIX operating system network code is derived from BSD, this probe is effectively intended to identify machines running UNIX.

Multicast addresses are legal only as destination addresses. The next rule pair denies and logs spoofed multicast network packets:

```
# Refuse Class D multicast addresses
# illegal only as a source address
# Multicast uses UDP
ipchains -A input  -i $EXTERNAL_INTERFACE \
         -s $CLASS_D_MULTICAST -j DENY -l
ipchains -A output -i $EXTERNAL_INTERFACE \
         -s $CLASS_D_MULTICAST -j REJECT -l
```

Legitimate multicast packets are always UDP packets. As such, multicast messages are sent point-to-point, just as any other UDP message is. The difference between unicast and multicast packets is the class of destination address used. The next rule denies outgoing multicast packets from your machine:

```
ipchains -A output -i $EXTERNAL_INTERFACE \
         -d $CLASS_D_MULTICAST -j REJECT -l
```

Multicast functionality is a configurable option when you compile the kernel, and your network interface card can be initialized to recognize multicast addresses. The functionality is enabled by default in Red Hat 6.0, but not in earlier releases. You might want to enable these addresses if you subscribe to a network conferencing service that provides multicast audio and video broadcasts.

You won't see legitimate multicast destination addresses unless you've registered yourself as a recipient. Multicast packets are sent to multiple, but specific, targets by prior arrangement. I have seen multicast packets sent out from machines on my ISP's local subnet, however. You can block multicast addresses altogether if you don't subscribe to multicast services. The next rule denies incoming multicast packets:

```
ipchains -A input  -i $EXTERNAL_INTERFACE -d $CLASS_D_MULTICAST -j REJECT -l
```

**Clarification on the Meaning of IP Address** *0.0.0.0*

Address `0.0.0.0` is reserved for use as a broadcast source address. The IPFW convention of specifying a match on any address, `any/0`, `0.0.0.0/0`, or `0.0.0.0/0.0.0.0`, doesn't match the broadcast source address. The reason is that a broadcast packet has a bit set in the packet header indicating that it's a broadcast packet destined for all interfaces on the network, rather than a point-to-point, unicast packet destined for a particular destination. Broadcast packets are handled differently from nonbroadcast packets. There is no legitimate nonbroadcast IP address `0.0.0.0`.

Multicast registration and routing is a complicated process managed by its own IP layer control protocol, the Internet Group Management Protocol (IGMP, protocol 2). For more information on multicast communication, refer to an excellent white paper, "How IP Multicast Works," by Vicki Johnson and Marjory Johnson. The paper is available at `http://www.ipmulticast.com/community/whitepapers/howipmcworks.html`.

Class D IP addresses range from `224.0.0.0` to `239.255.255.255`. The `CLASS_D_MULTICAST` constant, `224.0.0.0/4`, is defined to match on the first four bits of the address. As shown in Figure 3.2, in binary, the decimal values 224 (11100000B) to 239 (11101111B) are identical through the first four bits (1110B).

The next rule in this section denies and logs packets claiming to be from a Class E reserved network:

```
# Refuse Class E reserved IP addresses
ipchains -A input  -i $EXTERNAL_INTERFACE \
         -s $CLASS_E_RESERVED_NET -j DENY -l
```

Class E IP addresses range from `240.0.0.0` to `247.255.255.255`. The `CLASS_E_RESERVED_NET` constant, `240.0.0.0/5`, is defined to match on the first five bits of the address. As shown in Figure 3.3, in binary, the decimal values 240 (11110000B) to 247 (11110111B) are identical through the first five bits (1111 0B).

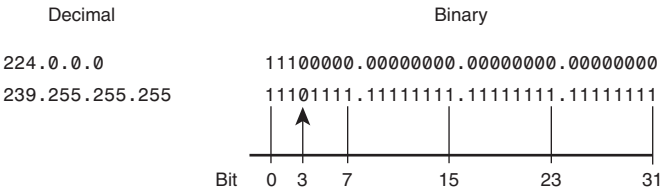


**Figure 3.2** The first matching four bits in the masked Class D multicast address range `224.0.0.0/4`.

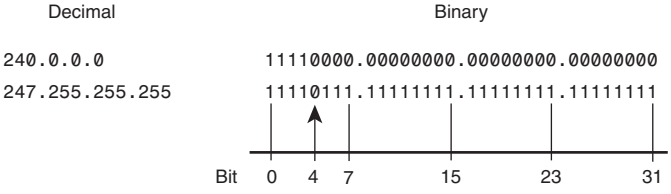


**Figure 3.3** The first matching five bits in the masked Class E reserved address range `240.0.0.0/5`.

The IANA ultimately manages the allocation and registration of the world's IP address space. For more information on IP address assignments, see `http://www.isi.edu/in-notes/iana/assignments/ipv4-address-space`. Some blocks of addresses are defined as reserved by the IANA. These addresses should not appear on the public Internet. The final set of rules deny this class of potentially spoofed packets:

```
# refuse addresses defined as reserved by the IANA
# 0.*.*.*, 1.*.*.*, 2.*.*.*, 5.*.*.*, 7.*.*.*, 23.*.*.*, 27.*.*.*
# 31.*.*.*, 37.*.*.*, 39.*.*.*, 41.*.*.*, 42.*.*.*, 58-60.*.*.*

ipchains -A input -i $EXTERNAL_INTERFACE -s 1.0.0.0/8 -j DENY -l
ipchains -A input -i $EXTERNAL_INTERFACE -s 2.0.0.0/8 -j DENY -l
ipchains -A input -i $EXTERNAL_INTERFACE -s 5.0.0.0/8 -j DENY -l
ipchains -A input -i $EXTERNAL_INTERFACE -s 7.0.0.0/8 -j DENY -l
ipchains -A input -i $EXTERNAL_INTERFACE -s 23.0.0.0/8 -j DENY -l
ipchains -A input -i $EXTERNAL_INTERFACE -s 27.0.0.0/8 -j DENY -l
ipchains -A input -i $EXTERNAL_INTERFACE -s 31.0.0.0/8 -j DENY -l
ipchains -A input -i $EXTERNAL_INTERFACE -s 37.0.0.0/8 -j DENY -l
ipchains -A input -i $EXTERNAL_INTERFACE -s 39.0.0.0/8 -j DENY -l
ipchains -A input -i $EXTERNAL_INTERFACE -s 41.0.0.0/8 -j DENY -l
ipchains -A input -i $EXTERNAL_INTERFACE -s 42.0.0.0/8 -j DENY -l
ipchains -A input -i $EXTERNAL_INTERFACE -s 58.0.0.0/7 -j DENY -l
ipchains -A input -i $EXTERNAL_INTERFACE -s 60.0.0.0/8 -j DENY -l

# 65: 01000001   - /3 includes 64 - need 65-79 spelled out
ipchains -A input -i $EXTERNAL_INTERFACE -s 65.0.0.0/8 -j DENY -l
ipchains -A input -i $EXTERNAL_INTERFACE -s 66.0.0.0/8 -j DENY -l
ipchains -A input -i $EXTERNAL_INTERFACE -s 67.0.0.0/8 -j DENY -l
ipchains -A input -i $EXTERNAL_INTERFACE -s 68.0.0.0/8 -j DENY -l
ipchains -A input -i $EXTERNAL_INTERFACE -s 69.0.0.0/8 -j DENY -l
ipchains -A input -i $EXTERNAL_INTERFACE -s 70.0.0.0/8 -j DENY -l
ipchains -A input -i $EXTERNAL_INTERFACE -s 71.0.0.0/8 -j DENY -l
ipchains -A input -i $EXTERNAL_INTERFACE -s 72.0.0.0/8 -j DENY -l
ipchains -A input -i $EXTERNAL_INTERFACE -s 73.0.0.0/8 -j DENY -l
ipchains -A input -i $EXTERNAL_INTERFACE -s 74.0.0.0/8 -j DENY -l
ipchains -A input -i $EXTERNAL_INTERFACE -s 75.0.0.0/8 -j DENY -l
ipchains -A input -i $EXTERNAL_INTERFACE -s 76.0.0.0/8 -j DENY -l
ipchains -A input -i $EXTERNAL_INTERFACE -s 77.0.0.0/8 -j DENY -l
ipchains -A input -i $EXTERNAL_INTERFACE -s 78.0.0.0/8 -j DENY -l
ipchains -A input -i $EXTERNAL_INTERFACE -s 79.0.0.0/8 -j DENY -l

# 80: 01010000   - /4 masks 80-95
ipchains -A input -i $EXTERNAL_INTERFACE -s 80.0.0.0/4 -j DENY -l

# 96: 01100000  - /4 masks 96-111
ipchains -A input -i $EXTERNAL_INTERFACE -s 96.0.0.0/4 -j DENY -l
```

```
# 126: 01111110  - /3 includes 127 - need 112-126 spelled out
ipchains -A input -i $EXTERNAL_INTERFACE -s 112.0.0.0/8 -j DENY -l
ipchains -A input -i $EXTERNAL_INTERFACE -s 113.0.0.0/8 -j DENY -l
ipchains -A input -i $EXTERNAL_INTERFACE -s 114.0.0.0/8 -j DENY -l
ipchains -A input -i $EXTERNAL_INTERFACE -s 115.0.0.0/8 -j DENY -l
ipchains -A input -i $EXTERNAL_INTERFACE -s 116.0.0.0/8 -j DENY -l
ipchains -A input -i $EXTERNAL_INTERFACE -s 117.0.0.0/8 -j DENY -l
ipchains -A input -i $EXTERNAL_INTERFACE -s 118.0.0.0/8 -j DENY -l
ipchains -A input -i $EXTERNAL_INTERFACE -s 119.0.0.0/8 -j DENY -l
ipchains -A input -i $EXTERNAL_INTERFACE -s 120.0.0.0/8 -j DENY -l
ipchains -A input -i $EXTERNAL_INTERFACE -s 121.0.0.0/8 -j DENY -l
ipchains -A input -i $EXTERNAL_INTERFACE -s 122.0.0.0/8 -j DENY -l
ipchains -A input -i $EXTERNAL_INTERFACE -s 123.0.0.0/8 -j DENY -l
ipchains -A input -i $EXTERNAL_INTERFACE -s 124.0.0.0/8 -j DENY -l
ipchains -A input -i $EXTERNAL_INTERFACE -s 125.0.0.0/8 -j DENY -l
ipchains -A input -i $EXTERNAL_INTERFACE -s 126.0.0.0/8 -j DENY -l

# 217: 11011001  - /5 includes 216 - need 217-219 spelled out
ipchains -A input -i $EXTERNAL_INTERFACE -s 217.0.0.0/8 -j DENY -l
ipchains -A input -i $EXTERNAL_INTERFACE -s 218.0.0.0/8 -j DENY -l
ipchains -A input -i $EXTERNAL_INTERFACE -s 219.0.0.0/8 -j DENY -l

# 223: 11011111  - /6 masks 220-223
ipchains -A input -i $EXTERNAL_INTERFACE -s 220.0.0.0/6 -j DENY -l
```

# Filtering ICMP Control and Status Messages

ICMP control messages are generated in response to a number of error conditions, and they are produced by network analysis programs, such as `ping` and `traceroute`. Table 3.2 lists the common ICMP message types of most interest to a small site.

**Table 3.2 Common ICMP Message Types**

| Type Code | Symbolic Name | Description |
|---|---|---|
| `0` | `echo-reply` | A `ping` response. |
| `3` | `destination-unreachable` | A general error status message; a router along the path to the destination is unable to deliver the packet to its next destination; used by `traceroute`. |
| `4` | `source-quench` | IP network layer flow control between two routers, or between a router and a host. |

*continues*

**Table 3.2 Continued**

| Type Code | Symbolic Name | Description |
|---|---|---|
| 5 | `redirect` | A routing message returned to the sender when a router determines that a shorter path exists. |
| 8 | `echo-request` | A `ping` request. |
| 11 | `time-exceeded` | A routing message returned when a packet's maximum hop count (TTL) is exceeded; used by `traceroute`. |
| 12 | `parameter-problem` | Unexpected values that are found in the IP packet header. |

## Error Status and Control Messages

Four ICMP control and status messages need to pass through the firewall: Source Quench, Parameter Problem, incoming Destination Unreachable, and outgoing Destination Unreachable, subtype Fragmentation Needed. Four other ICMP message types are optional: Echo Request, Echo Reply, other outgoing Destination Unreachable subtypes, and Time Exceeded. Other message types can be ignored, to be filtered out by the default policy.

Of the message types that can—or should—be ignored, only Redirect is listed in Table 3.2 because of its role in denial-of-service attacks as a Redirect bomb. (See Chapter 2 for more information on Redirect bombs.) As with Redirect, the remaining ICMP message types are specialized control and status messages intended for use between routers.

**ICMP Code Differences Between *ipchains* and *ipfwadm***

`ipchains` in Red Hat 6.0 supports the use of either the ICMP numeric message type or the alphabetic symbolic name. Earlier releases using `ipfwadm` supported only the numeric message type.

`ipchains` also supports use of the message subtypes, or codes. This is especially useful for finer filtering control over type 3 `destination-unreachable` messages. For example, you could specifically disallow outgoing `port-unreachable` messages to disable an incoming `traceroute`, or specifically allow only outgoing `Fragmentation Needed` status messages. `ipfwadm` does not support the message subtype codes.

To see a list of all supported ICMP symbolic names in `ipchains`, run `ipchains -h icmp`. To see the official RFC assignments, go to `http://www.isi.edu/in-notes/iana/assignments/icmp-parameters`.

The following sections describe the message types important to an endpoint host machine, as opposed to an intermediate router, in more detail.

### Source Quench Control (Type 4) Messages

ICMP message type 4, Source Quench, is sent when a connection source, usually a router, is sending data faster than the next destination router can handle it. Source Quench is used as a primitive form of flow control at the IP network layer, usually between two adjacent, point-to-point machines:

```
ipchains -A input  -i $EXTERNAL_INTERFACE -p icmp \
         -s $ANYWHERE 4 -d $IPADDR -j ACCEPT

ipchains -A output -i $EXTERNAL_INTERFACE -p icmp \
         -s $IPADDR 4 -d $ANYWHERE -j ACCEPT
```

The router's next hop or destination machine sends a Source Quench command. The originating router responds by sending packets at a slower rate, gradually increasing the rate until it receives another Source Quench message.

### Parameter Problem Status (Type 12) Messages

ICMP message type 12, Parameter Problem, is sent when a packet is received containing illegal or unexpected data in the header, or when the header checksum doesn't match the checksum generated by the receiving machine:

```
ipchains -A input  -i $EXTERNAL_INTERFACE -p icmp \
         -s $ANYWHERE 12 -d $IPADDR -j ACCEPT

ipchains -A output -i $EXTERNAL_INTERFACE -p icmp \
         -s $IPADDR 12 -d $ANYWHERE -j ACCEPT
```

### Destination Unreachable Error (Type 3) Messages

ICMP message type 3, Destination Unreachable, is a general error status message:

```
ipchains -A input  -i $EXTERNAL_INTERFACE -p icmp \
         -s $ANYWHERE 3 -d $IPADDR -j ACCEPT

ipchains -A output -i $EXTERNAL_INTERFACE -p icmp \
         -s $IPADDR 3 -d $ANYWHERE -j ACCEPT
```

The ICMP packet header for type 3, Destination Unreachable, messages contains an error code field identifying the particular kind of error. Ideally, you'd want to drop outgoing type 3 messages. This message type is what is sent in response to a hacker's attempt to map your service ports or address space. An attacker can create a denial-of-service condition by forcing your system to generate large numbers of these messages by bombarding your unused UDP ports. Worse, an attacker can spoof the source address, forcing your system to send them to the spoofed hosts. Unfortunately, the Destination Unreachable message creates a Catch-22 situation. One of the message subtypes, Fragmentation Needed, is used to negotiate packet fragment size. Your network performance can be seriously degraded without this negotiation.

If you want to respond to incoming `traceroute` requests, you must allow outgoing ICMP Destination Unreachable messages, subtype code Port Unreachable.

### Time Exceeded Status (Type 11) Messages

ICMP message type 11, Time Exceeded, indicates a timeout condition, or more accurately, that a packet's maximum hop count has been exceeded. On networks today, incoming Time Exceeded is mostly seen as the ICMP response to an outgoing UDP `traceroute` request:

```
ipchains -A input  -i $EXTERNAL_INTERFACE -p icmp \
         -s $ANYWHERE 11 -d $IPADDR -j ACCEPT

ipchains -A output -i $EXTERNAL_INTERFACE -p icmp \
         -s $IPADDR 11 -d $MY_ISP -j ACCEPT
```

If you want to respond to incoming `traceroute` requests, you must allow outgoing ICMP Time Exceeded messages. In the previous rules, only `traceroute`s from your ISP's machines are allowed. If you want to use `traceroute` yourself, you must allow incoming ICMP Time Exceeded messages. Because your machine is not an intermediate router, you have no other use for Time Exceeded messages.

## *ping* Echo Request (Type 8) and Echo Reply (Type 0) Control Messages

`ping` uses two ICMP message types. The request message, Echo Request, is message type 8. The reply message, Echo Reply, is message type 0. `ping` is a simple network analysis tool dating back to the original DARPANET. The name `ping` was taken from the idea of the audible ping played back by sonar systems. (DARPA is the Defense Advanced Research Projects Agency, after all.) Similar to sonar, an Echo Request message broadcast to all machines in a network address space generates Echo Reply messages, in return, from all hosts responding on the network.

> ***ipchains* Only: Using the ICMP Message Subtype Codes**
>
> For `ipchains` users only: The general, `ipfwadm`-compatible ruleset mentioned previously could be replaced with more specific rules allowing any outgoing type 3 messages to your ISP, and Fragmentation Needed messages to any address:
>
> ```
> ipchains -A input  -i $EXTERNAL_INTERFACE -p icmp \
>          -s $ANYWHERE 3 -d $IPADDR -j ACCEPT
>
> ipchains -A output -i $EXTERNAL_INTERFACE -p icmp \
>          -s $IPADDR 3 -d $MY_ISP -j ACCEPT
>
> ipchains -A output -i $EXTERNAL_INTERFACE -p icmp \
>          -s $IPADDR fragmentation-needed -d $ANYWHERE -j ACCEPT
> ```

### Outgoing *ping* to Remote Hosts

The following rule pair allows you to `ping` any host on the Internet:

```
# allow outgoing pings to anywhere
ipchains -A output -i $EXTERNAL_INTERFACE -p icmp \
         -s $IPADDR 8 -d $ANYWHERE -j ACCEPT

ipchains -A input  -i $EXTERNAL_INTERFACE -p icmp \
         -s $ANYWHERE 0 -d $IPADDR -j ACCEPT
```

### Incoming *ping* from Remote Hosts

The approach shown here allows only selected external hosts to `ping` you:

```
# allow incoming pings from trusted hosts
ipchains -A input  -i $EXTERNAL_INTERFACE -p icmp \
         -s $MY_ISP 8 -d $IPADDR -j ACCEPT

ipchains -A output -i $EXTERNAL_INTERFACE -p icmp \
         -s $IPADDR 0 -d $MY_ISP -j ACCEPT
```

For the purposes of example, external hosts allowed to `ping` your machine are machines belonging to your ISP. Chances are good that your network operations center or customer support will want to `ping` your external interface. Other than from your local network neighbors, other incoming Echo Requests are denied. `ping` is used in several different types of denial-of-service attacks.

### Blocking Incoming and Outgoing *smurf* Attacks

`smurf` attacks have used `ping` packets historically, continually broadcasting Echo Request messages to intermediate hosts with the source address spoofed to be the intended victim's IP address. As a result, every machine in the intermediary's network continually bombards the victim machine with Echo Reply messages, choking off all available bandwidth.

The following rule logs `smurf` attacks. Because the broadcast ICMP packets are not explicitly allowed, the firewall's deny-everything-by-default policy drops these packets anyway. Notice that all ICMP message types are denied, rather than just Echo Request messages. `ping` packets are typically used in `smurf` attacks, but other ICMP message types can be used as well. You can never be too careful in a firewall rule set:

```
# smurf attack
ipchains -A input  -i $EXTERNAL_INTERFACE -p icmp \
         -d $BROADCAST_DEST -j DENY -l

ipchains -A output -i $EXTERNAL_INTERFACE -p icmp \
         -d $BROADCAST_DEST -j REJECT -l

# smurf attack - network mask
ipchains -A input -i $EXTERNAL_INTERFACE -p icmp\
         -d $NETMASK -j DENY -l

ipchains -A output -i $EXTERNAL_INTERFACE -p icmp\
         -d $NETMASK -j REJECT -l
```

```
#smurf attack - network address
ipchains -A input -i $EXTERNAL_INTERFACE -p icmp\
         -d $NETWORK -j DENY -l
ipchains -A output -i $EXTERNAL_INTERFACE -p icmp\
         -d $NETWORK -j REJECT -l
```

## Protecting Services on Assigned Unprivileged Ports

LAN services, in particular, often run on unprivileged ports. For TCP-based services, a connection attempt to one of these services can be distinguished from an ongoing connection with a client using one of these unprivileged ports through the state of the `SYN` and `ACK` bits. You should block incoming connection attempts to these ports for your own security protection. You want to block outgoing connection attempts to protect yourself and others from mistakes on your end, and to log potential internal security problems.

What kinds of mistakes might you need protection from? The worst mistake is offering dangerous services to the world, whether inadvertently or intentionally, and is discussed in Chapter 2. A common mistake is running local network services that leak out to the Internet and bother other people. Another is allowing questionable outgoing traffic, such as port scans, whether this traffic is generated by accident or intentionally sent out by someone on your machine. A deny-everything-by-default firewall policy protects you from most mistakes of these types.

A deny-everything-by-default firewall policy allows you to run many private services behind the firewall without undo risk. These services must explicitly be allowed through the firewall to be accessible to remote clients. This generalization is only an approximation of reality, however. Although TCP services on privileged ports are reasonably safe from all but a skilled and determined hacker, UDP services are inherently less secure, and some services are assigned to run on unprivileged ports. RPC services, usually run over UDP, are even more problematic. RPC-based services are bound to some port, often an unprivileged port. The `portmap` daemon maps between the RPC service number and the actual port number. A port scan can show where these RPC-based services are bound without going through the `portmap` daemon.

### *Smurf* Attacks

Don't broadcast anything out unto the Internet. The `ping` broadcast mentioned previously is the basis of the `smurf` IP denial-of-service attack. See CERT Advisory `CA-98.01.smurf` at `www.cert.org` for more information on `smurf` attacks.

### Official Service Port Number Assignments

Port numbers are assigned and registered by the IANA. The information was originally maintained as RFC 1700, "Assigned Numbers." That RFC is now obsolete. The official information is dynamically maintained by the IANA at `http://www.isi.edu/in-notes/iana/assignments/port-numbers`.

## Common Local TCP Services Assigned to Unprivileged Ports

Some services, usually LAN services, are offered through an officially registered, well-known unprivileged port. Additionally, some services, such as FTP and IRC, use complicated communication protocols that don't lend themselves well to packet filtering. The rules described in the following sections disallow local or remote client programs from initiating a connection to one of these ports.

FTP is a good example of how the deny-by-default policy isn't always enough to cover all the possible cases. The FTP protocol is covered later in this chapter. For now, the important idea is that FTP allows connections between two unprivileged ports. Because some services listen on registered unprivileged ports, and the incoming connection request to these services are originating from an unprivileged client port, the rules allowing FTP inadvertantly allow incoming connections to these other, local, services, as well. This situation is also an example of how firewall rules are logically hierarchial and order-dependent. The rules explicitly protecting a LAN service running on an unprivileged port must precede the FTP rules allowing access to the entire unprivileged port range.

As a result, some of these rules appear to be redundant, and will be redundant for some people. For other people running other services, the following rules are necessary to protect private services running on local unprivileged ports.

### Disallowing Open Window Connections (TCP Port 2000)

Outgoing client connections to a remote Open Window manager should not be allowed. By specifying the `-y` flag, indicating the `SYN` bit, only a connection establishment attempt made from your machine is rejected. Other in-progress connections with remote client programs that are using the unprivileged port 2000 are not affected by the rule, because remote unprivileged ports are the endpoint of a connection initiated by a remote client to a server on your machine.

The following rule blocks local clients from initiating a connection request to a remote Open Window manager:

```
OPENWINDOWS_PORT="2000"                 # (TCP) OpenWindows

# Open Windows: establishing a connection
ipchains -A output -i $EXTERNAL_INTERFACE -p tcp -y \
         -s $IPADDR \
         -d $ANYWHERE $OPENWINDOWS_PORT -j REJECT
```

Incoming connections to port 2000 don't need to be explicitly blocked. Linux is not distributed with the Open Window manager.

> **The Problem with Port Scans**
>
> Port scans are not harmful in themselves. They're generated by network analysis tools. The problem with port scans today is that they are usually generated by people with less-than-honorable intentions. They are "analyzing" your network, not their own. Unfortunately, this leaves the merely curious looking guilty as well.

### Disallowing X Window Connections (TCP Ports 6000:6063)

Connections to remote X Window servers should be made over SSH, which automatically supports X Window connections. By specifying the `-y` flag, indicating the `SYN` bit, only connection establishment to the remote server port is being rejected. Other connections initiated using the port as a client port are not affected.

X Window port assignment begins at port 6000 with the first running server. If additional servers are run, each is assigned to the next incremental port. As a small site, you'll probably run a single X server, so your server will only listen on port 6000. Port 6063 is the highest assigned port, allowing 64 separate X Window managers running on a single machine:

```
XWINDOW_PORTS="6000:6063"               # (TCP) X Window
```

The first rule ensures that no outgoing connection attempts to remote X Window managers are made from your machine:

```
# X Window: establishing a remote connection
ipchains -A output -i $EXTERNAL_INTERFACE -p tcp -y \
         -s $IPADDR \
         -d $ANYWHERE $XWINDOW_PORTS -j REJECT
```

The next rule logs and blocks incoming connection attempts to your X Window manager. Local connections are not affected because local connections are made over the loopback interface:

```
# X Window: incoming connection attempt
ipchains -A input -i $EXTERNAL_INTERFACE -p tcp -y \
         -d $IPADDR $XWINDOW_PORTS -j DENY -l
```

### Disallowing SOCKS Server Connections (TCP Port 1080)

SOCKS is a local proxy server freely available from `http://www.socks.nec.com/`. Your SOCKS-aware client programs connect to the server instead of directly connecting to remote servers. The SOCKS server connects to remote servers, as a client, on your behalf.

Attempts to connect to remote SOCKS servers are fairly common and often involve intrusion exploits. The following rules allow port 1080 as a local or remote client port, but disallow port 1080 as a local or remote server port:

```
SOCKS_PORT="1080"                       # (TCP) socks
```

The first rule ensures that no outgoing connection attempts to remote SOCKS servers are made from your machine:

```
# SOCKS: establishing a connection
ipchains -A output -i $EXTERNAL_INTERFACE -p tcp -y \
         -s $IPADDR \
         -d $ANYWHERE $SOCKS_PORT -j REJECT -l
```

The next rule blocks incoming connection attempts to your SOCKS server:

```
# SOCKS: incoming connection
ipchains -A input -i $EXTERNAL_INTERFACE -p tcp -y \
         -d $IPADDR SOCKS_PORT -j DENY -l
```

## Common Local UDP Services Assigned to Unprivileged Ports

TCP protocol rules can be handled more precisely than UDP protocol rules, due to TCP's connection establishment protocol. As a datagram service, UDP doesn't have a connection state associated with it. Access to UDP services should simply be blocked. Explicit exceptions are made to accommodate DNS and any of the few other UDP-based Internet services you might use. Fortunately, the common UDP Internet services are often the type that are used between a client and a specific server. The filtering rules can often allow exchanges with one specific remote host.

NFS is the main UDP service to be concerned with. NFS runs on unprivileged port 2049. Unlike the previous TCP-based services, NFS is primarily a UDP-based service. It can be configured to run as a TCP-based service, but usually isn't.

### Disallowing NFS (UDP/TCP Port 2049) Connections

The first rule blocks NFS UDP port 2049 from any incoming access. The rule is unnecessary if you aren't running NFS. You shouldn't be running NFS on a firewall machine, but if you are, external access is denied:

```
NFS_PORT="2049"                         # (TCP/UDP) NFS

# NFS: UDP connections
ipchains -A input -i $EXTERNAL_INTERFACE -p udp \
         -d $IPADDR $NFS_PORT -j DENY -l
```

The next two TCP rules cover the little-used NFS TCP connection mode. Both incoming and outgoing connection establishment attempts are blocked, just as in the previous TCP sections:

```
# NFS: TCP connections
ipchains -A input -i $EXTERNAL_INTERFACE -p tcp -y \
         -d $IPADDR $NFS_PORT -j DENY -l

ipchains -A output -i $EXTERNAL_INTERFACE -p tcp -y \
         -d $ANYWHERE $NFS_PORT -j DENY -l
```

#### The TCP and UDP Service Protocol Tables

The remainder of this chapter is devoted to defining rules to allow access to specific services. Client/server communication, both for TCP- and UDP-based services, involves some kind of two-way communication using a protocol specific to the service. As such, access rules are always represented as an I/O pair. The client program makes a query, and the server sends a response. Rules for a given service are categorized as client rules or server rules. The client category represents the communication required for your local clients to access remote servers. The server category represents the communication required for remote clients to access the services hosted from your machines.

The application messages are encapsulated in either TCP or UDP transport protocol messages. Because each service uses an application protocol specific to itself, the particular characteristics of the TCP or UDP exchange is, to some extent, unique to the given service.

The exchange between client and server is explicitly described by the firewall rules. Part of the firewall rules' purpose is to ensure protocol integrity at the packet level. Firewall rules, expressed in `ipchains` syntax, are not overly human-readable, however. In each of the following sections, the service protocol at the packet-filtering level is presented as a table of state information, followed by the `ipchains` rules expressing those states.

Each row in the table lists a packet type involved in the service exchange. A firewall rule is defined for each individual packet type. The table is divided into columns:

- `Description` contains a brief description of whether the packet is originating from the client or the server, and the packet's purpose.
- `Protocol` is the transport protocol in use, TCP or UDP, or the IP protocol's control messages, ICMP.
- `Remote Address` is the legal address, or range of addresses, the packet can contain in the remote address field.
- `Remote Port` is the legal port, or range of ports, the packet can contain in the remote port field.
- `In/Out` describes the packet's direction, that is, whether it is coming into the system from a remote location or whether it is going out from the system to a remote location.
- `Local Address` is the legal address, or range of addresses, the packet can contain in the local address field.
- `Local Port` is the legal port, or range of ports, the packet can contain in the local port field.
- TCP protocol packets contain a final column, TCP Flag, defining the legal `SYN–ACK` states the packet may have.

The table describes packets as either incoming or outgoing. Addresses and ports are described as either remote or local, relative to your machine's network interface. Notice that for incoming packets, remote address and port refer to the source fields in the IP packet header. Local address and port refer to the destination fields in the IP packet header. For outgoing packets, remote address and port refer to the destination fields in the IP packet header. Local address and port refer to the source fields in the IP packet header.

Finally, in the few instances where the service protocol involves ICMP messages, notice that the IP network layer ICMP packets are not associated with the concept of a source or destination port, as is the case for transport layer TCP or UDP packets. Instead, ICMP packets use the concept of a control or status message type. ICMP messages are not sent to programs bound to particular service ports. Instead, ICMP messages are sent from one computer to another. Consequently, the few ICMP packet entries presented in the tables use the source port column to contain the message type. For incoming ICMP packets, the source port column is the Remote Port column. For outgoing ICMP packets, the source port column is the Local Port column.

# Enabling Basic, Required Internet Services

Only two services are truly required: the Domain Name Service (DNS) and the `IDENT` user identification service. DNS translates between hostnames and their associated IP addresses. You can't locate a remote host without DNS. `identd` provides the username or ID associated with a connection. This is commonly requested by a remote mail server when you send email. You don't need to offer `identd` service, but you must account for incoming connection requests in some way to avoid lengthy timeouts.

## Allowing DNS (UDP/TCP Port 53)

DNS uses a communication protocol that relies on both UDP and TCP. Connection modes include regular client-to-server connections, peer-to-peer traffic between forwarding servers and full servers, and primary and secondary name server connections.

Query lookup requests are normally done over UDP, both for client-to-server lookups and peer-to-peer server lookups. The UDP communication can fail for a client-to-server lookup if the information being returned is too large to fit in a single UDP DNS packet. The server sets a flag bit in the DNS message header indicating that the data is truncated. In this case, the protocol allows for a retry over TCP. Figure 3.4 shows the relationship between UDP and TCP during a DNS lookup. In practice, TCP isn't normally needed for queries. TCP is conventionally used for administrative zone transfers between primary and secondary name servers.

Zone transfers are the transfer of a name server's complete information about a network, or the piece (zone) of a network, the server is authoritative for (i.e., the official server). The authoritative name server is referred to as the primary name server. Secondary, or backup, name servers periodically request zone transfers from their primary to keep their DNS caches up-to-date.

For example, one of your ISP's name servers is the primary, authoritative server for the ISP's LAN address space. ISPs often have multiple DNS servers to balance the load, as well as for backup redundancy. The other name servers are secondary name servers, refreshing their information from the master copy on the primary server.

Zone transfers are beyond the scope of this book. A small system isn't likely to be an authoritative name server for a public domain's name space, nor is it likely to be a public backup server for that information.

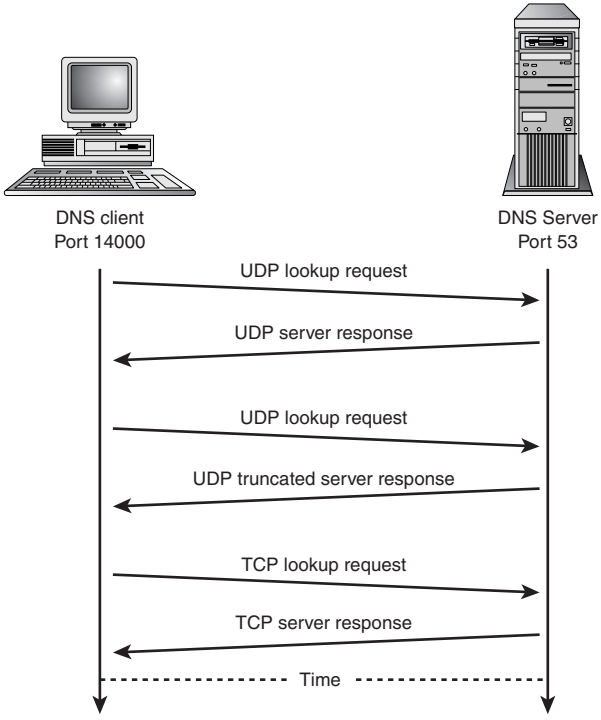


**Figure 3.4** DNS client-to-server lookup.

Table 3.3 lists the complete DNS protocol for which the firewall rules account.

**Table 3.3 DNS Protocol**

| Description | Protocol | Remote Address | Remote Port | In/ Out | Local Address | Local Port | TCP Flag |
|---|---|---|---|---|---|---|---|
| Local client query | UDP | NAMESERVER | 53 | Out | IPADDR | 1024:65535 | — |
| Remote server response | UDP | NAMESERVER | 53 | In | IPADDR | 1024:65535 | — |
| Local client query | TCP | NAMESERVER | 53 | Out | IPADDR | 1024:65535 | Any |
| Remote server response | TCP | NAMESERVER | 53 | In | IPADDR | 1024:65535 | Ack |
| Local server query | UDP | NAMESERVER | 53 | Out | IPADDR | 53 | — |
| Remote server response | UDP | NAMESERVER | 53 | In | IPADDR | 53 | — |
| Local zone transfer request | TCP | Primary | 53 | Out | IPADDR | 1024:65535 | Any |
| Remote zone transfer request | TCP | Primary | 53 | In | IPADDR | 1024:65535 | ACK |
| Remote client query | UDP | DNS client | 1024:65535 | In | IPADDR | 53 | — |
| Local server response | UDP | DNS client | 1024:65535 | Out | IPADDR | 53 | — |
| Remote client query | TCP | DNS client | 1024:65535 | In | IPADDR | 53 | Any |
| Local server response | TCP | DNS client | 1024:65535 | Out | IPADDR | 53 | ACK |
| Remote client query | UDP | DNS client | 53 | In | IPADDR | 53 | — |
| Local server response | UDP | DNS client | 53 | Out | IPADDR | 53 | — |
| Remote zone transfer request | TCP | Secondary | 1024:65535 | In | IPADDR | 53 | Any |
| Local zone transfer response | TCP | Secondary | 1024:65535 | Out | IPADDR | 53 | ACK |

### Allowing DNS Lookups as a Client

The DNS resolver client isn't a specific program. The client is incorporated into the network library code compiled into network programs. When a hostname requires a lookup, the resolver requests the lookup from a `named` server. Many small systems are configured only as a DNS client. The server runs on a remote machine. For a home user, the name server is usually a machine owned by your ISP.

DNS sends a lookup request as a UDP datagram:

```
NAMESERVER ="my.name.server"            # (TCP/UDP) DNS

ipchains -A output -i $EXTERNAL_INTERFACE -p udp \
         -s $IPADDR $UNPRIVPORTS \
         -d $NAMESERVER 53 -j ACCEPT

ipchains -A input  -i $EXTERNAL_INTERFACE -p udp \
         -s $NAMESERVER 53 \
         -d $IPADDR $UNPRIVPORTS -j ACCEPT
```

If an error occurs because the returned data is too large to fit in a UDP datagram, DNS retries using a TCP connection.

The next two rules are included for the rare occasion when the lookup response won't fit in a DNS UDP datagram. They won't be used in normal, day-to-day operations. You could run your system without problem for months on end without the TCP rules. Unfortunately, every so often—perhaps once or twice a year—your DNS lookups hang without these rules due to a poorly configured remote DNS server. More typically, these rules are used by a secondary name server requesting a zone transfer from its primary name server:

```
ipchains -A output -i $EXTERNAL_INTERFACE -p tcp \
         -s $IPADDR $UNPRIVPORTS \
         -d <my.dns.primary> 53 -j ACCEPT

ipchains -A input -i $EXTERNAL_INTERFACE -p tcp ! -y \
         -s <my.dns.primary> 53 \
         -d $IPADDR $UNPRIVPORTS -j ACCEPT
```

### Allowing Your DNS Lookups as a Peer-to-Peer, Forwarding Server

Peer-to-peer transactions are exchanges between two servers. In the case of DNS, when your local name server doesn't have the information a client requested stored locally, it contacts a remote server and forwards the request to it.

Configuring a local forwarding name server can be a big performance gain. As shown in Figure 3.5, when `named` is configured as a caching and forwarding name server, it functions both as a local server and as a client to a remote DNS server. The difference between a direct client-to-server exchange and a forwarded local server–to–remote server exchange (peer-to-peer) is in the source and destination ports used. Instead of initiating an exchange from an unprivileged port, `named` initiates the exchange from its own DNS port 53. A second difference is that peer-to-peer server lookups of this type are always done over UDP.

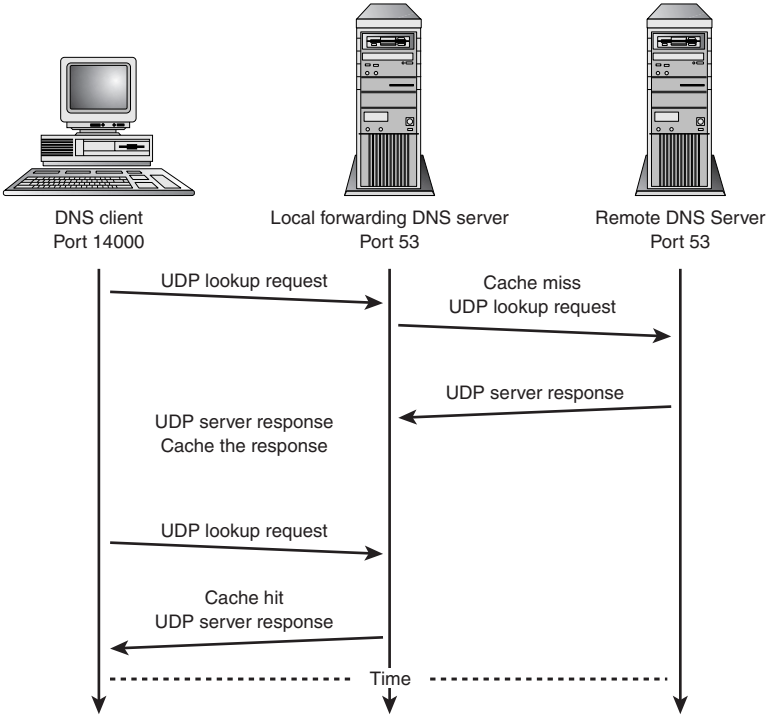


**Figure 3.5** A DNS forwarding server and peer-to-peer lookup.

Local client requests are sent to the local DNS server. The first time, `named` won't have the lookup information, so it forwards the request to a remote name server. `named` caches the returned information and passes it on to the client. The next time the same information is requested, `named` finds it in its local cache and doesn't make a remote request:

```
ipchains -A output -i $EXTERNAL_INTERFACE -p udp \
         -s $IPADDR 53 \
         -d $NAMESERVER 53 -j ACCEPT

ipchains -A input  -i $EXTERNAL_INTERFACE -p udp \
         -s $NAMESERVER 53 \
         -d $IPADDR 53 -j ACCEPT
```

### Allowing Remote DNS Lookups to Your Server

The average home-based site has little reason to provide DNS service to remote machines. Unless your site is an ISP, your clients will be machines local to your LAN, your subnet, your router, your network address space, or whatever terms are in use for the size of network and organization you're looking at.

Assuming you are a home or small business offering DNS to the outside world, you would limit the clients to a select group. You would not allow connections from just anywhere:

```
# client-to-server DNS transaction
ipchains -A input  -i $EXTERNAL_INTERFACE -p udp \
         -s <my.dns.clients> $UNPRIVPORTS \
         -d $IPADDR 53 -j ACCEPT

ipchains -A output -i $EXTERNAL_INTERFACE -p udp \
         -s $IPADDR 53 \
         -d <my.dns.clients> $UNPRIVPORTS -j ACCEPT

# peer-to-peer server DNS transaction
ipchains -A input  -i $EXTERNAL_INTERFACE -p udp \
         -s <my.dns.clients> 53 \
         -d $IPADDR 53 -j ACCEPT

ipchains -A output -i $EXTERNAL_INTERFACE -p udp \
         -s $IPADDR 53 \
         -d <my.dns.clients> 53 -j ACCEPT
```

The next two rules apply to client request retries when the data is too large to fit in a UDP DNS packet. They also apply to a secondary name server requesting zone transfers from a primary name server. TCP is used almost exclusively for zone transfers, and zone transfers represent a number of potential security holes. If used, it's important to limit the client source addresses:

```
ipchains -A input  -i $EXTERNAL_INTERFACE -p tcp \
         -s <my.dns.secondaries> $UNPRIVPORTS \
         -d $IPADDR 53 -j ACCEPT

ipchains -A output -i $EXTERNAL_INTERFACE -p tcp ! -y \
         -s $IPADDR 53 \
         -d <my.dns.secondaries> $UNPRIVPORTS -j ACCEPT
```

### *Filtering the* AUTH *User Identification Service (TCP Port 113)*

The `AUTH`, or `IDENTD`, user identification service is most often used when sending mail or posting a Usenet article. Some FTP sites are also configured to require a resolvable `AUTH` lookup. For logging purposes, the server initiates an `AUTH` request back to your machine to get the account name of the user who initiated the mail or news connection. Table 3.4 lists the complete client/server connection protocol for the `AUTH` service.

> **DNS Zone Transfers over TCP**
>
> Large-scale network services, such as DNS zone transfers, should not be allowed by small sites. Undoubtedly, someone somewhere is an exception. For the exceptions, and for those individuals who are "going to do it anyway," limit the list of secondaries you accept connections from. Share your DNS tables with only trusted remote sites.

**Table 3.4** ***identd*** **Protocol**

| Description | Protocol | Remote Address | Remote Port | In/ Out | Local Address | Local Port | TCP Flag |
|---|---|---|---|---|---|---|---|
| Local client query | TCP | ANYWHERE | 113 | Out | IPADDR | 1024:65535 | Any |
| Remote server response | TCP | ANYWHERE | 113 | In | IPADDR | 1024:65535 | Ack |
| Remote client query | TCP | ANYWHERE | 1024:65535 | In | IPADDR | 113 | Any |
| Local server response | TCP | ANYWHERE | 1024:65535 | Out | IPADDR | 113 | Ack |

### *Allowing Your Outgoing* AUTH *Requests as a Client*

Your machine would act as an AUTH client if you ran a mail or FTP server. There is no reason not to allow your system to be an AUTH client:

```
ipchains -A output -i $EXTERNAL_INTERFACE -p tcp \
         -s $IPADDR $UNPRIVPORTS \
         -d $ANYWHERE 113 -j ACCEPT

ipchains -A input -i $EXTERNAL_INTERFACE -p tcp ! -y \
         -s $ANYWHERE 113 \
         -d $IPADDR $UNPRIVPORTS -j ACCEPT
```

### *Filtering Incoming* AUTH *Requests to Your Server*

Offering AUTH services is the subject of ongoing debate. There appear to be no overwhelming arguments to make the case for either side, other than that a few FTP sites require it, and AUTH provides user account information. Whether you decide to offer the service or not, you will receive incoming requests for the service every time you send mail.

If you run the identd server out of /etc/inetd.conf, the following rules enable incoming identd connection requests:

```
ipchains -A input  -i $EXTERNAL_INTERFACE -p tcp \
         -s $ANYWHERE $UNPRIVPORTS \
         -d $IPADDR 113 -j ACCEPT

ipchains -A output -i $EXTERNAL_INTERFACE -p tcp ! -y \
         -s $IPADDR 113 \
         -d $ANYWHERE $UNPRIVPORTS -j ACCEPT
```

If you decide not to offer the service, you can't just deny the incoming requests. The result would be a long wait each time you tried to send mail or post a Usenet

article. Your mail client won't be notified that the mail or article was received for delivery until the `identd` request timed out. Instead, you need to reject the connection request to avoid waiting for the TCP connection timeout. This is the only case where an incoming packet is rejected rather than denied in these examples:

```
ipchains -A input -i $EXTERNAL_INTERFACE -p tcp \
         -s $ANYWHERE \
         -d $IPADDR 113 -j REJECT
```

# Enabling Common TCP Services

Possibly no one will want to enable all the services listed in this section, but everyone will want to enable some subset of them. These are the services most often used over the Internet today. As such, this section is more of a reference section than anything else. This section provides rules for the following:

- Email
- Usenet
- `telnet`
- `ssh`
- `ftp`
- Web services
- `finger`
- `whois`
- `gopher` Information Service
- Wide Area Information Service (WAIS)

Many other services are available that aren't covered here. Some of them are used on specialized servers, some by large businesses and organizations, and some are designed for use in local, private networks.

## Email (TCP SMTP Port 25, POP Port 110, IMAP Port 143)

Email is a service almost everyone wants. How mail is set up depends on your ISP, your connection type, and your own choices. Email is sent across the network using the SMTP protocol assigned to TCP service port 25. Email is commonly received locally through one of three different protocols—SMTP, POP, or IMAP—depending on the services your ISP provides and on your local configuration.

SMTP is the general mail protocol. Mail is delivered to the destination host machine. The endpoint mail server determines whether the mail is deliverable (addressed to a valid user account on the machine) and delivers it to the user's local mailbox.

POP and IMAP are mail retrieval services. POP runs on TCP port 110. IMAP runs on TCP port 143. ISPs commonly make incoming mail available to their customers using one of these two services. Both services are authenticated. They are associated with the ISP customer's user account and password. As far as mail retrieval is concerned, the difference between SMTP and POP or IMAP is that SMTP receives incoming mail and queues it in the user's local mailbox. POP and IMAP retrieve mail into the user's local mail program from the user's ISP, where the mail had been queued remotely in the user's SMTP mailbox at the ISP. Table 3.5 lists the complete client/server connection protocols for SMTP, POP, and IMAP.

**Table 3.5 SMTP, POP, and IMAP Mail Protocols**

| Description | Protocol | Remote Address | Remote Port | In/ Out | Local Address | Local Port | TCP Flag |
|---|---|---|---|---|---|---|---|
| Send outgoing mail | TCP | ANYWHERE | 25 | Out | IPADDR | 1024:65535 | Any |
| Remote server response | TCP | ANYWHERE | 25 | In | IPADDR | 1024:65535 | Ack |
| Receive incoming mail | TCP | ANYWHERE | 1024:65535 | In | IPADDR | 25 | Any |
| Local server response | TCP | ANYWHERE | 1024:65536 | Out | IPADDR | 25 | Ack |
| Local client query | TCP | POP SERVER | 110 | Out | IPADDR | 1024:65535 | Any |
| Remote server response | TCP | POP SERVER | 110 | In | IPADDR | 1024:65535 | Ack |
| Remote client query | TCP | POP client | 1024:65535 | In | IPADDR | 110 | Any |
| Local server response | TCP | POP client | 1024:65535 | Out | IPADDR | 110 | ACK |
| Local client query | TCP | IMAP SERVER | 143 | Out | IPADDR | 1024:65535 | Any |
| Remote server response | TCP | IMAP SERVER | 143 | In | IPADDR | 1024:65535 | ACK |
| Remote client query | TCP | IMAP client | 1024:65535 | In | IPADDR | 143 | Any |
| Local server response | TCP | IMAP client | 1024:65535 | Out | IPADDR | 143 | ACK |

### Sending Mail Over SMTP (TCP Port 25)

Mail is sent over SMTP. But whose SMTP server do you use to collect your mail and send it onward? ISPs offer SMTP mail service to their customers. The ISP's mail server acts as the mail gateway. It knows how to collect your mail, find the recipient host, and relay the mail. With UNIX, you can host your own local mail server if you want. Your server will be responsible for routing the mail to its destination.

#### *Relaying Outgoing Mail Through an External (ISP) Gateway SMTP Server*

When you relay outgoing mail through an external gateway SMTP server, your client mail program sends all outgoing mail to your ISP's mail server. Your ISP acts as your mail gateway to the rest of the world. Your system doesn't need to know how to locate your mail destinations or the routes to them. The ISP mail gateway serves as your relay.

The following two rules allow you to relay mail through your ISP's SMTP gateway:

```
SMTP_GATEWAY="my.isp.server"          # external mail server or relay

ipchains -A output -i $EXTERNAL_INTERFACE -p tcp \
         -s $IPADDR $UNPRIVPORTS \
         -d $SMTP_GATEWAY 25 -j ACCEPT

ipchains -A input -i $EXTERNAL_INTERFACE -p tcp ! -y \
         -s $SMTP_GATEWAY 25 \
         -d $IPADDR $UNPRIVPORTS -j ACCEPT
```

#### *Sending Mail to Any External Mail Server*

Alternatively, you can bypass your ISP's mail server and host your own. Your local server is responsible for collecting your outgoing mail, doing the DNS lookup on the destination hostname, and relaying the mail to its destination. Your client mail program points to your local SMTP server rather than to the ISP's server.

The following two rules allow you to send mail directly to the remote destinations:

```
ipchains -A output -i $EXTERNAL_INTERFACE -p tcp \
         -s $IPADDR $UNPRIVPORTS \
         -d $ANYWHERE 25 -j ACCEPT

ipchains -A input -i $EXTERNAL_INTERFACE -p tcp ! -y \
         -s $ANYWHERE 25 \
         -d $IPADDR $UNPRIVPORTS -j ACCEPT
```

**Proxy Servers As Both Client and Server**

The current SMTP mail server is `sendmail`, which is a proxy server. It acts as a server to the client mail program sending the mail. It acts as a client to the remote server it's sending the mail to. The terms *client* and *server* can be confusing in this context. `sendmail` acts as both, depending on which program it's talking to.

### Receiving Mail

How you receive mail depends on your situation. If you run your own local mail server, you can collect incoming mail directly on your Linux machine. If you retrieve your mail from your ISP account, you may or may not retrieve mail as a POP or IMAP client, depending on how you've configured your ISP email account, and depending on the mail delivery services the ISP offers.

#### *Receiving Mail as a Local SMTP Server (TCP Port 25)*

If you want to receive mail sent directly to your local machines from anywhere in the world, you need to run `sendmail` and use these server rules:

```
ipchains -A input  -i $EXTERNAL_INTERFACE -p tcp \
         -s $ANYWHERE $UNPRIVPORTS \
         -d $IPADDR 25 -j ACCEPT

ipchains -A output -i $EXTERNAL_INTERFACE -p tcp ! -y \
         -s $IPADDR 25 \
         -d $ANYWHERE $UNPRIVPORTS -j ACCEPT
```

Alternatively, if you'd rather keep your local email account relatively private and use your work or ISP email account as your public address, you could configure your work and ISP mail accounts to forward mail to your local server. In this case, you could replace the previous single rule pair, accepting connections from anywhere, with separate, specific rules for each mail forwarder.

#### *Retrieving Mail as a POP Client (TCP Port 110)*

Connecting to a POP server is a very common means of retrieving mail from a remote ISP or work account. If your ISP uses a POP server for customer mail retrieval, you need to allow outgoing client-to-server connections.

The server's address will be a specific hostname or address, rather than the global `ANYWHERE` specifier. POP accounts are user accounts associated with a specific user and password:

```
POP_SERVER="my.isp.pop.server"          # external pop server, if any

ipchains -A output -i $EXTERNAL_INTERFACE -p tcp \
         -s $IPADDR $UNPRIVPORTS \
         -d $POP_SERVER 110 -j ACCEPT

ipchains -A input -i $EXTERNAL_INTERFACE -p tcp ! -y \
         -s $POP_SERVER 110 \
         -d $IPADDR $UNPRIVPORTS -j ACCEPT
```

#### *Receiving Mail as an IMAP Client (TCP Port 143)*

Connecting to an IMAP server is another common means of retrieving mail from a remote ISP or work account. If your ISP uses an IMAP server for customer mail retrieval, you need to allow outgoing client-to-server connections.

The server's address will be a specific hostname or address, rather than the global ANYWHERE specifier. IMAP accounts are user accounts associated with a specific user and password:

```
IMAP_SERVER="my.isp.imap.server"       # external imap server, if any

ipchains -A output -i $EXTERNAL_INTERFACE -p tcp \
         -s $IPADDR $UNPRIVPORTS \
         -d $IMAP_SERVER 143 -j ACCEPT

ipchains -A input -i $EXTERNAL_INTERFACE -p tcp ! -y \
         -s $IMAP_SERVER 143 \
         -d $IPADDR $UNPRIVPORTS -j ACCEPT
```

### Examples of Real-World Client and Server Email Combinations

Four common approaches to client and server email combinations are described in this section:

- Sending mail as an SMTP client and receiving mail as a POP client
- Sending mail as an SMTP client and receiving mail as an IMAP client
- Sending mail as an SMTP client and receiving mail as an SMTP server
- Sending mail as an SMTP server and receiving mail as an SMTP server

The first two are useful if you rely completely on your ISP's SMTP and POP or IMAP email services. The third example is a mixed approach, relaying outgoing mail through your ISP's SMTP mail server, but receiving mail directly through your local SMTP server. The fourth approach supports running your own complete, independent mail server for both outgoing and incoming mail.

#### *Sending Mail as an SMTP Client and Receiving Mail as a POP Client*

If you are sending mail as an SMTP client and receiving mail as a POP client, you are relying completely on a remote site for your mail services. The remote site hosts both an SMTP server for relaying your outgoing mail, and a POP server for local mail retrieval:

```
SMTP_GATEWAY="my.isp.server"          # external mail server or relay

ipchains -A output -i $EXTERNAL_INTERFACE -p tcp \
         -s $IPADDR $UNPRIVPORTS \
         -d $SMTP_GATEWAY 25 -j ACCEPT

ipchains -A input -i $EXTERNAL_INTERFACE -p tcp ! -y \
         -s $SMTP_GATEWAY 25 \
         -d $IPADDR $UNPRIVPORTS -j ACCEPT

POP_SERVER="my.isp.pop.server"        # external pop server, if any
```

```
ipchains -A output -i $EXTERNAL_INTERFACE -p tcp \
         -s $IPADDR $UNPRIVPORTS \
         -d $POP_SERVER 110 -j ACCEPT

ipchains -A input -i $EXTERNAL_INTERFACE -p tcp ! -y \
         -s $POP_SERVER 110 \
         -d $IPADDR $UNPRIVPORTS -j ACCEPT
```

### *Sending Mail as an SMTP Client and Receiving Mail as an IMAP Client*

If you are sending mail as an SMTP client and receiving mail as an IMAP client, you are relying completely on a remote site for your mail services. The remote site hosts both an SMTP server for relaying outgoing mail and an IMAP server for local mail retrieval:

```
SMTP_GATEWAY="my.isp.server"          # external mail server or relay

ipchains -A output -i $EXTERNAL_INTERFACE -p tcp \
         -s $IPADDR $UNPRIVPORTS \
         -d $SMTP_GATEWAY 25 -j ACCEPT

ipchains -A input -i $EXTERNAL_INTERFACE -p tcp ! -y \
         -s $SMTP_GATEWAY 25 \
         -d $IPADDR $UNPRIVPORTS -j ACCEPT

IMAP_SERVER="my.isp.imap.server"       # external imap server, if any

ipchains -A output -i $EXTERNAL_INTERFACE -p tcp \
         -s $IPADDR $UNPRIVPORTS \
         -d $IMAP_SERVER 143 -j ACCEPT
ipchains -A input -i $EXTERNAL_INTERFACE -p tcp ! -y \
         -s $IMAP_SERVER 143 \
         -d $IPADDR $UNPRIVPORTS -j ACCEPT
```

### *Sending Mail as an SMTP Client and Receiving Mail as an SMTP Server*

If you are sending mail as an SMTP client and receiving mail as an SMTP server, you are relying on a remote site to offer SMTP service to relay your outgoing mail to remote destinations. You run `sendmail` locally as a local SMTP server allowing remote hosts to send mail to your machine directly. Outgoing mail is relayed through your ISP, but the local `sendmail` daemon knows how to deliver incoming mail to local user accounts:

```
SMTP_GATEWAY="my.isp.server"            # external mail server or relay

ipchains -A output -i $EXTERNAL_INTERFACE -p tcp \
         -s $IPADDR $UNPRIVPORTS \
         -d $SMTP_GATEWAY 25 -j ACCEPT
```

```
ipchains -A input -i $EXTERNAL_INTERFACE -p tcp ! -y \
         -s $SMTP_GATEWAY 25 \
         -d $IPADDR $UNPRIVPORTS -j ACCEPT

ipchains -A input  -i $EXTERNAL_INTERFACE -p tcp \
         -s $ANYWHERE $UNPRIVPORTS \
         -d $IPADDR 25 -j ACCEPT

ipchains -A output -i $EXTERNAL_INTERFACE -p tcp ! -y \
         -s $IPADDR 25 \
         -d $ANYWHERE $UNPRIVPORTS -j ACCEPT
```

#### *Sending Mail as an SMTP Server and Receiving Mail as an SMTP Server*

If you are sending mail as an SMTP server and receiving mail as an SMTP server, you provide all your own mail services. Your local `sendmail` daemon is configured to relay outgoing mail to the destination hosts itself, as well as collect and deliver incoming mail:

```
ipchains -A output -i $EXTERNAL_INTERFACE -p tcp \
         -s $IPADDR $UNPRIVPORTS \
         -d $ANYWHERE 25 -j ACCEPT

ipchains -A input -i $EXTERNAL_INTERFACE -p tcp ! -y \
         -s $ANYWHERE 25 \
         -d $IPADDR $UNPRIVPORTS -j ACCEPT

ipchains -A input  -i $EXTERNAL_INTERFACE -p tcp \
         -s $ANYWHERE $UNPRIVPORTS \
         -d $IPADDR 25 -j ACCEPT

ipchains -A output -i $EXTERNAL_INTERFACE -p tcp ! -y \
         -s $IPADDR 25 \
         -d $ANYWHERE $UNPRIVPORTS -j ACCEPT
```

### Hosting a Mail Server for Remote Clients

Hosting public POP or IMAP services is unusual for a small system. You might do this if you offered remote mail services to a few friends, for example, or if their ISP mail service was temporarily unavailable. In any case, it's important to limit the clients your system will accept connections from, both on the packet-filtering level and on the server configuration level. You should also consider using an encrypted authentication method, or allow mail retrieval only over an SSH connection.

#### *Hosting a POP Server for Remote Clients*

POP servers are one of the three most common and successful points of entry for hacking exploits.

If you use a local system as a central mail server and run a local `popd` server to provide mail access to local machines on a LAN, you don't need the server rules in this example. Incoming connections from the Internet should be denied. If you do need to

host POP service for a limited number of remote individuals, the next two rules allow incoming connections to your POP server. Connections are limited to your specific clients' IP addresses:

```
ipchains -A input  -i $EXTERNAL_INTERFACE -p tcp \
         -s <my.pop.clients> $UNPRIVPORTS \
         -d $IPADDR 110 -j ACCEPT

ipchains -A output -i $EXTERNAL_INTERFACE -p tcp ! -y \
         -s $IPADDR 110 \
         -d <my.pop.clients> $UNPRIVPORTS -j ACCEPT
```

### *Hosting an IMAP Server for Remote Clients*

IMAP servers are one of the three most common and successful points of entry for hacking exploits.

If you use a local system as a central mail server and run a local `imapd` server to provide mail access to local machines on a LAN, you don't need a server rule. Incoming connections from the Internet should be denied. If you do need to host IMAP service for a limited number of remote individuals, the next two rules allow incoming connections to your IMAP server. Connections are limited to your specific clients' IP addresses:

```
ipchains -A input  -i $EXTERNAL_INTERFACE -p tcp \
         -s <my.imap.clients> $UNPRIVPORTS \
         -d $IPADDR 143 -j ACCEPT

ipchains -A output -i $EXTERNAL_INTERFACE -p tcp ! -y \
         -s $IPADDR 143 \
         -d <my.imap.clients> $UNPRIVPORTS -j ACCEPT
```

## Accessing Usenet News Services (TCP NNTP Port 119)

Usenet news is accessed over NNTP running on top of TCP through service port 119. Reading news and posting articles are handled by your local news client. Few systems require the server rules. Table 3.6 lists the complete client/server connection protocol for the NNTP Usenet news service.

**Table 3.6 NNTP Protocol**

| Description | Protocol | Remote Address | Remote Port | In/ Out | Local Address | Local Port | TCP Flag |
|---|---|---|---|---|---|---|---|
| Local client query | TCP | NEWS SERVER | 119 | Out | IPADDR | 1024:65535 | Any |
| Remote server response | TCP | NEWS SERVER | 119 | In | IPADDR | 1024:65535 | Ack |
| Remote client query | TCP | NNTP clients | 1024:65535 | In | IPADDR | 119 | Any |
| Local server response | TCP | NNTP clients | 1024:65535 | Out | IPADDR | 119 | Ack |
| Local server query | TCP | News feed | 119 | Out | IPADDR | 1024:65535 | Any |
| Remote server response | TCP | News feed | 119 | In | IPADDR | 1024:65535 | Ack |

### Reading and Posting News as a Usenet Client

The client rules allow connections to your ISP's news server. Both reading news and posting articles are handled by these rules:

```
NEWS_SERVER="my.news.server"           # external news server, if any

ipchains -A output -i $EXTERNAL_INTERFACE -p tcp \
         -s $IPADDR $UNPRIVPORTS \
         -d $NEWS_SERVER 119 -j ACCEPT

ipchains -A input -i $EXTERNAL_INTERFACE -p tcp ! -y \
         -s $NEWS_SERVER 119 \
         -d $IPADDR $UNPRIVPORTS -j ACCEPT
```

### Hosting a Usenet News Server for Remote Clients

A small site is very unlikely to host a news server for the outside world. Even hosting a local news server is unlikely. For the rare exception, the server rules should be configured to allow incoming connections only from a select set of clients:

```
ipchains -A input  -i $EXTERNAL_INTERFACE -p tcp \
         -s <my.news.clients> $UNPRIVPORTS \
         -d $IPADDR 119 -j ACCEPT

ipchains -A output -i $EXTERNAL_INTERFACE -p tcp ! -y \
         -s $IPADDR 119 \
         -d <my.news.clients> $UNPRIVPORTS -j ACCEPT
```

### Allowing Peer News Feeds for a Local Usenet Server

A small, home-based site is unlikely to have a peer-to-peer news feed server relationship with an ISP. Although news servers used to be fairly accessible to the general Internet, few open news servers are available anymore due to SPAM and server load issues.

If your site is large enough or rich enough to host a general Usenet server, you have to get your news feed from somewhere. The next two rules allow your local news server to receive its news feed from a remote server. The local server contacts the remote server as a client. The only difference between the peer-to-peer news feed rules and the regular client rules is the name or address of the remote host:

```
ipchains -A output -i $EXTERNAL_INTERFACE -p tcp \
         -s $IPADDR $UNPRIVPORTS \
         -d <my.news.feed> 119 -j ACCEPT

ipchains -A input -i $EXTERNAL_INTERFACE -p tcp ! -y \
         -s <my.news.feed> 119 \
         -d $IPADDR $UNPRIVPORTS -j ACCEPT
```

## *telnet* (TCP Port 23)

`telnet` has been the de facto standard means of remote login over the Internet for many years. As the nature of the Internet community has changed, `telnet` has come to be viewed more and more as an insecure service, because it communicates in ASCII clear text. Nevertheless, `telnet` may be the only tool available to you for remote connections, depending on the connection options available at the other end. If you have the option, you should always use an encrypted service, such as `ssh`, rather than `telnet`.

The client and server rules here allow access to and from anywhere. If you use `telnet`, you can probably limit the external addresses to a select subset at the packet-filtering level. Table 3.7 lists the complete client/server connection protocol for the TELNET service.

**Table 3.7 TELNET Protocol**

| Description | Protocol | Remote Address | Remote Port | In/ Out | Local Address | Local Port | TCP Flag |
|---|---|---|---|---|---|---|---|
| Local client request | TCP | ANYWHERE | 23 | Out | IPADDR | 1024:65535 | Any |
| Remote server response | TCP | ANYWHERE | 23 | In | IPADDR | 1024:65535 | Ack |
| Remote client request | TCP | `telnet` clients | 1024:65535 | In | IPADDR | 23 | Any |
| Local server response | TCP | `telnet` clients | 1024:65535 | Out | IPADDR | 23 | Ack |

### Allowing Outgoing Client Access to Remote Sites

If you need to use `telnet` to access your accounts on remote systems, the next two rules allow outgoing connections to remote sites. If your site has multiple users, you might limit outgoing connections to the specific sites your users have accounts on, rather than allowing outgoing connections to `anywhere`:

```
ipchains -A output -i $EXTERNAL_INTERFACE -p tcp \
         -s $IPADDR $UNPRIVPORTS \
         -d $ANYWHERE 23 -j ACCEPT

ipchains -A input -i $EXTERNAL_INTERFACE -p tcp ! -y \
         -s $ANYWHERE 23 \
         -d $IPADDR $UNPRIVPORTS -j ACCEPT
```

### Allowing Incoming Access to Your Local Server

Even if you need client access to remote servers, you may not need to allow incoming connections to your TELNET server. If you do, the next two rules allow incoming connections to your server:

```
ipchains -A input  -i $EXTERNAL_INTERFACE -p tcp \
         -s $ANYWHERE $UNPRIVPORTS \
         -d $IPADDR 23 -j ACCEPT

ipchains -A output -i $EXTERNAL_INTERFACE -p tcp ! -y \
         -s $IPADDR 23 \
         -d $ANYWHERE $UNPRIVPORTS -j ACCEPT
```

Rather than allow connections from anywhere, it is preferable to define server rules for each specific host or network an incoming connection can legitimately originate from.

## *ssh* (TCP Port 22)

SSH, secure shell, isn't included in Linux distributions due to export limitations on cryptographic technology, but it is freely available from software sites on the Internet. SSH is considered far preferable to using `telnet` for remote login access, because both ends of the connection use authentication keys for both hosts and users, and data is encrypted. Additionally, SSH is more than a remote login service. It can automatically direct X Window connections between remote sites, and FTP and other TCP-based connections can be directed over the more secure SSH connection. Provided that the other end of the connection allows SSH connections, it's possible to route all TCP connections through the firewall using SSH. As such, SSH is something of a poor man's virtual private network (VPN).

The ports used by SSH are highly configurable. The rules in this example apply to the default SSH port usage. By default, connections are initiated between a client's unprivileged port and the server's assigned service port 22. The server forks off a copy of itself for the connection, and the client end of the connection is then reassigned to

a privileged port in the descending range from 1023 to 513 in order to support `.rhosts` and `hosts.equiv` authentication. The first available port is used. The SSH client will optionally use the unprivileged ports exclusively. The SSH server will accept connections from either the privileged or unprivileged ports.

The client and server rules here allow access to and from anywhere. In practice, you would limit the external addresses to a select subset, particularly because both ends of the connection must be configured to recognize each individual user account for authentication. Table 3.8 lists the complete client/server connection protocol for the SSH service.

**Table 3.8 SSH Protocol**

| Description | Protocol | Remote Address | Remote Port | In/ Out | Local Address | Local Port | TCP Flag |
|---|---|---|---|---|---|---|---|
| Local client request | TCP | ANYWHERE | 22 | Out | IPADDR | 1024:65535 | Any |
| Remote server response | TCP | ANYWHERE | 22 | In | IPADDR | 1024:65535 | Ack |
| Local client request | TCP | ANYWHERE | 22 | Out | IPADDR | 513:1023 | Any |
| Remote server response | TCP | ANYWHERE | 22 | In | IPADDR | 513:1023 | Ack |
| Remote client request | TCP | SSH clients | 1024:65535 | In | IPADDR | 22 | Any |
| Local server response | TCP | SSH clients | 1024:65535 | Out | IPADDR | 22 | Ack |
| Remote client request | TCP | SSH clients | 513:1023 | In | IPADDR | 22 | Any |
| Local server response | TCP | SSH clients | 513:1023 | Out | IPADDR | 22 | Ack |

### SSH, *tcp_wrappers*, and *rhost* Authentication

SSH cannot be started under `tcp_wrappers` directly, but it can be compiled to honor the access list information in `/etc/hosts.allow` and `/etc/hosts.deny`.

`.rhosts` and `hosts.equiv` authentication should simply not be available on a firewall machine. System security analysis tools discussed in Chapter 8, "Intrusion Detection and Incident Reporting" warn you if these files exist on the system.

For more information on SSH, refer to `http://www.ssh.fi/`.

When selecting a privileged server port for the ongoing connection, the first free port between 1023 to 513 is used. The range of ports you allow equates to the number of simultaneous incoming SSH connections you allow:

```
SSH_PORTS="1020:1023"               # (TCP) 4 simultaneous connections
```

### Allowing Client Access to Remote SSH Servers

These rules allow you to connect to remote sites using `ssh`:

```
ipchains -A output -i $EXTERNAL_INTERFACE -p tcp \
         -s $IPADDR $UNPRIVPORTS \
         -d $ANYWHERE 22 -j ACCEPT

ipchains -A input -i $EXTERNAL_INTERFACE -p tcp ! -y \
         -s $ANYWHERE 22 \
         -d $IPADDR $UNPRIVPORTS -j ACCEPT

ipchains -A output -i $EXTERNAL_INTERFACE -p tcp \
         -s $IPADDR $SSH_PORTS \
         -d $ANYWHERE 22 -j ACCEPT

ipchains -A input  -i $EXTERNAL_INTERFACE -p tcp ! -y \
         -s $ANYWHERE 22 \
         -d $IPADDR $SSH_PORTS -j ACCEPT
```

### Allowing Remote Client Access to Your Local SSH Server

These rules allow incoming connections to your `sshd` server:

```
ipchains -A input  -i $EXTERNAL_INTERFACE -p tcp \
         -s $ANYWHERE $UNPRIVPORTS \
         -d $IPADDR 22 -j ACCEPT

ipchains -A output -i $EXTERNAL_INTERFACE -p tcp ! -y \
         -s $IPADDR 22 \
         -d $ANYWHERE $UNPRIVPORTS -j ACCEPT

ipchains -A input -i $EXTERNAL_INTERFACE -p tcp \
         -s $ANYWHERE $SSH_PORTS \
         -d $IPADDR 22 -j ACCEPT

ipchains -A output -i $EXTERNAL_INTERFACE -p tcp ! -y \
         -s $IPADDR 22 \
         -d $ANYWHERE $SSH_PORTS -j ACCEPT
```

## *ftp* (TCP Ports 21, 20)

FTP remains one of the most common means of transferring files between two networked machines. Web-based interfaces to FTP have become common, as well.

FTP uses two privileged ports, one for sending commands and one for sending data. Port 21 is used to establish the initial connection to the server and pass user

commands. Port 20 is used to establish a data channel over which files and directory listings are sent as data.

FTP has two modes for exchanging data between a client and server, normal data channel port mode and passive data channel mode. Normal port mode is the original, default mechanism when using the `ftp` client program and connecting to a remote FTP site. Passive mode is a newer mechanism, and is the default when connecting through a Web browser. Occasionally, you might encounter an FTP site that supports only one mode or the other. Table 3.9 lists the complete client/server connection protocol for the FTP service.

**Table 3.9 FTP Protocol**

| Description | Protocol | Remote Address | Remote Port | In/ Out | Local Address | Local Port | TCP Flag |
|---|---|---|---|---|---|---|---|
| Local client query | TCP | ANYWHERE | 21 | Out | IPADDR | 1024:65535 | Any |
| Remote server response | TCP | ANYWHERE | 21 | In | IPADDR | 1024:65535 | Ack |
| Remote server port data channel request | TCP | ANYWHERE | 20 | In | IPADDR | 1024:65535 | Any |
| Local client port data channel response | TCP | ANYWHERE | 20 | Out | IPADDR | 1024:65535 | Ack |
| Local client passive data channel request | TCP | ANYWHERE | 1024:65535 | Out | IPADDR | 1024:65535 | Any |
| Remote server passive data channel response | TCP | ANYWHERE | 1024:65535 | In | IPADDR | 1024:65535 | Ack |
| Remote client request | TCP | ANYWHERE | 1024:65535 | In | IPADDR | 21 | Any |
| Local server response | TCP | ANYWHERE | 1024:65535 | Out | IPADDR | 21 | ACK |
| Local server port data channel request | TCP | ANYWHERE | 1024:65535 | Out | IPADDR | 20 | Any |
| Remote client port data channel response | TCP | ANYWHERE | 1024:65535 | In | IPADDR | 20 | ACK |
| Remote client passive data channel request | TCP | ANYWHERE | 1024:65535 | In | IPADDR | 1024:65535 | Any |
| Local server passive data channel response | TCP | ANYWHERE | 1024:65535 | Out | IPADDR | 1024:65535 | ACK |

### Allowing Outgoing Client Access to Remote FTP Servers

It's almost a given that most sites will want FTP client access to remote file repositories. Most people will want to enable outgoing client connections to a remote server.

#### *Outgoing FTP Requests*

The next two rules allow an outgoing connection to a remote FTP server:

```
ipchains -A output -i $EXTERNAL_INTERFACE -p tcp \
         -s $IPADDR $UNPRIVPORTS \
         -d $ANYWHERE 21 -j ACCEPT

ipchains -A input -i $EXTERNAL_INTERFACE -p tcp ! -y \
         -s $ANYWHERE 21 \
         -d $IPADDR $UNPRIVPORTS -j ACCEPT
```

#### *Normal Port Mode FTP Data Channels*

The next two rules allow the standard data channel connection, where the remote server calls back to establish the data connection:

```
ipchains -A input  -i $EXTERNAL_INTERFACE -p tcp \
        -s $ANYWHERE 20 \
        -d $IPADDR $UNPRIVPORTS -j ACCEPT

ipchains -A output -i $EXTERNAL_INTERFACE -p tcp ! -y \
        -s $IPADDR $UNPRIVPORTS \
        -d $ANYWHERE 20 -j ACCEPT
```

This unusual callback behavior, where the remote server establishes the secondary connection with your client, is part of what makes FTP difficult to secure at the packet-filtering level. There is no mechanism to assure that the incoming connection is truly originating from the remote FTP server you've contacted. Unless you've explicitly blocked incoming connections to local services running on unprivileged ports, such as an X Window or SOCKS server, remote access to these services is allowed by the FTP client rules for port mode data channels.

#### *Passive Mode FTP Data Channels*

The next two rules allow the newer passive data channel mode used by Web browsers:

```
ipchains -A output -i $EXTERNAL_INTERFACE -p tcp \
        -s $IPADDR $UNPRIVPORTS \
        -d $ANYWHERE $UNPRIVPORTS -j ACCEPT

ipchains -A input  -i $EXTERNAL_INTERFACE -p tcp ! -y \
        -s $ANYWHERE $UNPRIVPORTS \
        -d $IPADDR $UNPIRVPORTS -j ACCEPT
```

Passive mode is considered more secure than port mode because the `ftp` client initiates both the control and data connections, even though the connection is made between two unprivileged ports.

### Allowing Incoming Access to Your Local FTP Server

Whether to offer FTP services to the world is a difficult decision. Although FTP sites abound on the Internet, FTP server configuration requires great care. Numerous FTP security exploits are possible.

If your goal is to offer general read-only access to some set of files on your machine, you might consider making these files available through a Web server. If your goal is to allow file uploads to your machine from the outside, FTP server access should be severely limited on the firewall level, on the `tcp_wrapper` level, and on the FTP configuration level.

In any case, if you decide to offer FTP services, and if you decide to allow incoming file transfers, write access should not be allowed via Anonymous FTP. Remote write access to your file systems should be allowed only from specific, authenticated FTP user accounts, from specific remote sites, and to carefully controlled and limited FTP areas reserved in your file system. Chapter 7, "Issues At the UNIX System Administration Level," discusses these FTP issues.

#### *Incoming FTP Requests*

The next two rules allow incoming connections to your FTP server:

```
ipchains -A input  -i $EXTERNAL_INTERFACE -p tcp \
         -s $ANYWHERE $UNPRIVPORTS \
         -d $IPADDR 21 -j ACCEPT

ipchains -A output -i $EXTERNAL_INTERFACE -p tcp ! -y \
         -s $IPADDR 21 \
         -d $ANYWHERE $UNPRIVPORTS -j ACCEPT
```

#### *Normal Port Mode FTP Data Channel Responses*

The next two rules allow the FTP server to call back the remote client and establish the secondary data channel connection:

```
ipchains -A output -i $EXTERNAL_INTERFACE -p tcp \
         -s $IPADDR 20 \
         -d $ANYWHERE $UNPRIVPORTS -j ACCEPT

ipchains -A input  -i $EXTERNAL_INTERFACE -p tcp ! -y \
         -s $ANYWHERE $UNPRIVPORTS \
         -d $IPADDR 20 -j ACCEPT
```

#### *Passive Mode FTP Data Channel Responses*

The next two rules allow the remote FTP client to establish the secondary data channel connection with the local server:

```
ipchains -A input  -i $EXTERNAL_INTERFACE -p tcp \
         -s $ANYWHERE $UNPRIVPORTS \
         -d $IPADDR $UNPRIVPORTS -j ACCEPT

ipchains -A output -i $EXTERNAL_INTERFACE -p tcp ! -y \
         -s $IPADDR $UNPRIVPORTS \
         -d $ANYWHERE $UNPRIVPORTS -j ACCEPT
```

## Web Services

Web services are based on Hypertext Transfer Protocol (HTTP). Client and server connections use the standard TCP conventions. Several higher-level, special-purpose communication protocols are available, in addition to the standard general HTTP access, including secure access over SSL, and access via an ISP-provided Web server proxy. These different access protocols use different service ports.

### Standard HTTP Access (TCP Port 80)

In normal use, Web services are available over `http` service port 80. Table 3.10 lists the complete client/server connection protocol for the HTTP Web service.

**Table 3.10 HTTP Protocol**

| Description | Protocol | Remote Address | Remote Port | In/ Out | Local Address | Local Port | TCP Flag |
|---|---|---|---|---|---|---|---|
| Local client request | TCP | ANYWHERE | 80 | Out | IPADDR | 1024:65535 | Any |
| Remote server response | TCP | ANYWHERE | 80 | In | IPADDR | 1024:65535 | Ack |
| Remote client request | TCP | ANYWHERE | 1024:65535 | In | IPADDR | 80 | Any |
| Local server response | TCP | ANYWHERE | 1024:65535 | Out | IPADDR | 80 | Ack |

#### *Accessing Remote Web Sites as a Client*

It's almost inconceivable in today's world that a home-based site would not want to access the World Wide Web from a Web browser. The next two rules allow access to remote Web servers:

```
ipchains -A output -i $EXTERNAL_INTERFACE -p tcp \
         -s $IPADDR $UNPRIVPORTS \
         -d $ANYWHERE 80 -j ACCEPT

ipchains -A input -i $EXTERNAL_INTERFACE -p tcp ! -y \
         -s $ANYWHERE 80 \
         -d $IPADDR $UNPRIVPORTS -j ACCEPT
```

**Caution: Don't Use *tftp* on the Internet**

`tftp` offers a simplified, unauthenticated, UDP version of the FTP service. It is intended for loading boot software into routers and diskless workstations over a local network from trusted hosts. Some people confuse `tftp` as an alternative to `ftp`. Don't use it over the Internet, period.

### *Allowing Remote Access to a Local Web Server*

If you decide to run a Web server of your own and host a Web site for the Internet, the general server rules allow all typical incoming access to your site. This is all most people need to host a Web site:

```
ipchains -A input -i $EXTERNAL_INTERFACE -p tcp\
         -s $ANYWHERE $UNPRIVPORTS \
         -d $IPADDR 80 -j ACCEPT

ipchains -A output -i $EXTERNAL_INTERFACE -p tcp ! -y \
         -s $IPADDR 80 \
         -d $ANYWHERE $UNPRIVPORTS -j ACCEPT
```

## Secure Web Access (SSL) (TCP Port 443)

Secure Socket Layer (SSL) is used for secure, encrypted Web access. The SSL protocol uses TCP port 443. You will most often encounter this if you go to a commercial Web site to purchase something, use online banking services, or enter a protected Web area where you'll be prompted for personal information. Table 3.11 lists the complete client/server connection protocol for the SSL service.

**Table 3.11 SSL Protocol**

| Description | Protocol | Remote Address | Remote Port | In/ Out | Local Address | Local Port | TCP Flag |
|---|---|---|---|---|---|---|---|
| Local client request | TCP | ANYWHERE | 443 | Out | IPADDR | 1024:65535 | Any |
| Remote server response | TCP | ANYWHERE | 443 | In | IPADDR | 1024:65535 | Ack |
| Remote client request | TCP | ANYWHERE | 1024:65535 | In | IPADDR | 443 | Any |
| Local server response | TCP | ANYWHERE | 1024:65535 | Out | IPADDR | 443 | Ack |

### *Accessing Remote Web Sites Over SSL as a Client*

Most people will want client access to secure Web sites at some point or another:

```
ipchains -A output -i $EXTERNAL_INTERFACE -p tcp \
         -s $IPADDR $UNPRIVPORTS \
         -d $ANYWHERE 443 -j ACCEPT

ipchains -A input -i $EXTERNAL_INTERFACE -p tcp ! -y \
         -s $ANYWHERE 443 \
         -d $IPADDR $UNPRIVPORTS -j ACCEPT
```

### *Allowing Remote Access to a Local SSL Web Server*

If you conduct some form of e-commerce, you'll mostly likely want to allow incoming connections to SSL-protected areas of your Web site. Otherwise, you won't need local server rules.

The basic Apache Web server distribution comes with SSL support, but the more secure SSL modules are not included due to Federal encryption regulations. Both free and commercial SSL support packages are available for the Apache Web server, however. See `www.apache.org` for more information.

The next two rules allow incoming access to your Web server using the SSL protocol:

```
ipchains -A input  -i $EXTERNAL_INTERFACE -p tcp \
         -s $ANYWHERE $UNPRIVPORTS \
         -d $IPADDR 443 -j ACCEPT

ipchains -A output -i $EXTERNAL_INTERFACE -p tcp ! -y \
         -s $IPADDR 443 \
         -d $ANYWHERE $UNPRIVPORTS-j ACCEPT
```

## Web Proxy Access (TCP Ports 8008, 8080)

Publicly accessible Web server proxies are most common at ISPs. As a customer, you configure your browser to use a remote proxy service. Web proxies are often accessed through one of two unprivileged ports assigned for this purpose, port 8008 or 8080, as defined by the ISP. In return, you get faster Web page access when the pages are already cached locally at your ISP's server, and the anonymity of proxied access to remote sites. Your connections are not direct, but instead are initiated on your behalf by your ISP's proxy. Table 3.12 lists the local client to remote server connection protocol for the Web proxy service.

**Table 3.12 Web Proxy Protocol**

| Description | Protocol | Remote Address | Remote Port | In/ Out | Local Address | Local Port | TCP Flag |
|---|---|---|---|---|---|---|---|
| Local client request | TCP | WEB PROXY SERVER | WEB PROXY PORT | Out | IPADDR | 1024:65535 | Any |
| Remote server response | TCP | WEB PROXY SERVER | WEB PROXY PORT | In | IPADDR | 1024:65535 | Ack |

If you use a Web proxy service offered by your ISP, the specific server address and port number will be defined by your ISP. The client rules are:

```
WEB_PROXY_SERVER="my.www.proxy"         # ISP Web proxy server, if any
WEB_PROXY_PORT="www.proxy.port"         # ISP Web proxy port, if any
                                        # typically 8008 or 8080

ipchains -A output -i $EXTERNAL_INTERFACE -p tcp \
         -s $IPADDR $UNPRIVPORTS \
         -d $WEB_PROXY_SERVER $WEB_PROXY_PORT -j ACCEPT

ipchains -A input -i $EXTERNAL_INTERFACE -p tcp ! -y \
         -s $WEB_PROXY_SERVER $WEB_PROXY_PORT \
         -d $IPADDR $UNPRIVPORTS -j ACCEPT
```

## *finger* (TCP Port 79)

From a connection point of view, the `finger` service is harmless. Due to the changing nature of privacy issues in relation to a growing and changing Internet community, offering `finger` service is generally discouraged today. `finger` provides user account information, including such things as user login name, real name, currently active logins, pending mail, and mail forwarding locations. Often, `finger` provides user-furnished (in a `.plan` file) personal information as well, including phone numbers, home addresses, tasks and plans, vacation status, and so forth. Table 3.13 lists the complete client/server connection protocol for the `finger` service.

**Table 3.13 *finger* Protocol**

| Description | Protocol | Remote Address | Remote Port | In/ Out | Local Address | Local Port | TCP Flag |
|---|---|---|---|---|---|---|---|
| Local client request | TCP | ANYWHERE | 79 | Out | IPADDR | 1024:65535 | Any |
| Remote server response | TCP | ANYWHERE | 79 | In | IPADDR | 1024:65535 | Ack |
| Remote client request | TCP | finger clients | 1024:65535 | In | IPADDR | 79 | Any |
| Local server response | TCP | finger clients | 1024:65535 | Out | IPADDR | 79 | Ack |

### Accessing Remote *finger* Servers as a Client

There's no harm in enabling outgoing access to remote `finger` servers, and these are the rules to do so:

```
ipchains -A output -i $EXTERNAL_INTERFACE -p tcp \
         -s $IPADDR $UNPRIVPORTS \
         -d $ANYWHERE 79 -j ACCEPT

ipchains -A input -i $EXTERNAL_INTERFACE -p tcp ! -y \
         -s $ANYWHERE 79 \
         -d $IPADDR $UNPRIVPORTS -j ACCEPT
```

### Allowing Remote Client Access to a Local *finger* Server

If you choose to allow outside access to your `finger` service, it's recommended that you limit access to specific client sites. `finger` access can be limited both at the firewall level and at the `tcp_wrapper` level.

The next rules allow incoming connections to your `finger` server, but only selected remote hosts are allowed to initiate the connection:

```
ipchains -A input  -i $EXTERNAL_INTERFACE -p tcp \
         -s <my.finger.clients> $UNPRIVPORTS \
         -d $IPADDR 79 -j ACCEPT

ipchains -A output -i $EXTERNAL_INTERFACE -p tcp ! -y \
         -s $IPADDR 79 \
         -d <my.finger.clients> $UNPRIVPORTS -j ACCEPT
```

## *whois* (TCP Port 43)

The `whois` program accesses the InterNIC Registration Services database. It allows IP address and host and domain name lookups in human-readable form. Table 3.14 lists the local client to remote server connection protocol for the `whois` service.

**Table 3.14 *whois* Protocol**

| Description | Protocol | Remote Address | Remote Port | In/ Out | Local Address | Local Port | TCP Flag |
|---|---|---|---|---|---|---|---|
| Local client request | TCP | ANYWHERE | 43 | Out | IPADDR | 1024:65535 | Any |
| Remote server response | TCP | ANYWHERE | 43 | In | IPADDR | 1024:65535 | Ack |

The next two rules allow you to query an official remote server:

```
ipchains -A output -i $EXTERNAL_INTERFACE -p tcp \
         -s $IPADDR $UNPRIVPORTS \
         -d $ANYWHERE 43 -j ACCEPT

ipchains -A input -i $EXTERNAL_INTERFACE -p tcp ! -y \
         -s $ANYWHERE 43 \
         -d $IPADDR $UNPRIVPORTS -j ACCEPT
```

## *gopher* (TCP Port 70)

The GOPHER information service is still available for low-overhead ASCII terminals, but its use has largely been replaced by Web-based search engines and hypertext links. It is unlikely that a Linux system would offer local GOPHER service instead of a Web site. Server rules are not included. Table 3.15 lists the local client to remote server connection protocol for the GOPHER service.

**Table 3.15 *gopher* Protocol**

| Description | Protocol | Remote Address | Remote Port | In/ Out | Local Address | Local Port | TCP Flag |
|---|---|---|---|---|---|---|---|
| Local client request | TCP | ANYWHERE | 70 | Out | IPADDR | 1024:65535 | Any |
| Remote server response | TCP | ANYWHERE | 70 | In | IPADDR | 1024:65535 | Ack |

The following are the client rules that allow you to connect to a remote server:

```
ipchains -A output -i $EXTERNAL_INTERFACE -p tcp \
         -s $IPADDR $UNPRIVPORTS \
         -d $ANYWHERE 70 -j ACCEPT

ipchains -A input -i $EXTERNAL_INTERFACE -p tcp ! -y \
         -s $ANYWHERE 70 \
         -d $IPADDR $UNPRIVPORTS -j ACCEPT
```

## WAIS (TCP Port 210)

Wide Area Information Servers (WAIS) are now known as search engines. Web browsers typically provide a graphical front-end to WAIS. Netscape contains the WAIS client code necessary to connect to WAIS. Table 3.16 lists the local client to remote server connection protocol for the WAIS service.

**Table 3.16 WAIS Protocol**

| Description | Protocol | Remote Address | Remote Port | In/ Out | Local Address | Local Port | TCP Flag |
|---|---|---|---|---|---|---|---|
| Local client request | TCP | ANYWHERE | 210 | Out | IPADDR | 1024:65535 | Any |
| Remote server response | TCP | ANYWHERE | 210 | In | IPADDR | 1024:65535 | Ack |

The following two rules allow client access to remote WAIS services:

```
ipchains -A output -i $EXTERNAL_INTERFACE -p tcp \
         -s $IPADDR $UNPRIVPORTS \
         -d $ANYWHERE 210 -j ACCEPT

ipchains -A input -i $EXTERNAL_INTERFACE -p tcp ! -y \
         -s $ANYWHERE 210 \
         -d $IPADDR $UNPRIVPORTS -j ACCEPT
```

# Enabling Common UDP Services

The stateless UDP protocol is inherently less secure than the connection-based TCP protocol. Because of this, many security-conscious sites completely disable, or else limit as much as possible, all access to UDP services. Obviously, UDP-based DNS exchanges are necessary, but the remote name servers can be explicitly specified in the firewall rules. As such, this section provides rules for only three services:

- `traceroute`
- Dynamic Host Configuration Protocol (DHCP)
- Network Time Protocol (NTP)

## *traceroute* (UDP Port 33434)

`traceroute` is a UDP service that causes intermediate systems to generate ICMP Time Exceeded messages to gather hop count information, and the target system to return a Destination Unreachable (port not found) message, indicating the endpoint of the route to the host. By default, the firewall being developed in this chapter blocks incoming UDP `traceroute` packets destined to the port range `traceroute` generally uses. As a result, outgoing ICMP responses to incoming `traceroute` requests won't be sent. Table 3.17 lists the complete client/server exchange protocol for the `traceroute` service.

**Table 3.17** ***traceroute*** **Protocol**

| Description | Protocol | Remote Address | Remote Port/ ICMP Type | In/ Out | Local Address | Local Port/ ICMP Typ |
|---|---|---|---|---|---|---|
| Outgoing `traceroute` probe | UDP | ANYWHERE | 33434:33523 | Out | IPADDR | 32769:65535 |
| Time Exceeded (intermediate hop) | ICMP | ANYWHERE | 11 | In | IPADDR | — |
| Port not found (termination) | ICMP | ANYWHERE | 3 | In | IPADDR | — |
| Incoming `traceroute` probe | UDP | ISP | 32769:65535 | In | IPADDR | 33434:33523 |
| Time Exceeded (intermediate hop) | ICMP | ISP | — | Out | IPADDR | 11 |
| Port not found (termination) | ICMP | ISP | — | Out | IPADDR | 3 |

`traceroute` can be configured to use any port or port range. As such, it's difficult to block all incoming `traceroute` packets by listing specific ports. However, it often uses source ports in the range from 32769 to 65535 and destination ports in the range from 33434 to 33523. Symbolic constants are defined for `traceroute`'s default source and destination ports:

```
TRACEROUTE_SRC_PORTS="32769:65535"
TRACEROUTE_DEST_PORTS="33434:33523"
```

### Enabling Outgoing *traceroute* Requests

If you intend to use `traceroute` yourself, you must enable the UDP client ports. Note that you must allow incoming ICMP Time Exceeded and Destination Unreachable messages from anywhere for outgoing `traceroute` to work:

```
ipchains -A output -i $EXTERNAL_INTERFACE -p udp \
         -s $IPADDR $TRACEROUTE_SRC_PORTS \
         -d $ANYWHERE $TRACEROUTE_DEST_PORTS -j ACCEPT
```

### Allowing Incoming *traceroute* Requests

Because `traceroute` is a less secure UDP service and can be used to attack other UDP services, the following example opens incoming `traceroute` from only your ISP and its associated Network Operations Center:

```
ipchains -A input -i $EXTERNAL_INTERFACE -p udp \
         -s $MY_ISP $TRACEROUTE_SRC_PORTS \
         -d $IPADDR $TRACEROUTE_DEST_PORTS -j ACCEPT
```

Note that you must allow outgoing ICMP Time Exceeded and Destination Unreachable messages to be targeted to your ISP for incoming `traceroute` to work.

## Accessing Your ISP's DHCP Server (UDP Ports 67, 68)

DHCP exchanges, if any, between your site and your ISP's server will necessarily be local client to remote server exchanges. DHCP clients receive temporary, dynamically allocated IP addresses from a central server that manages the ISP's customer IP address space.

If you have a dynamically allocated IP address from your ISP, you need to run the DHCP client daemon (either `dhcpd` or `pump`) on your machine. It's not uncommon for bogus DHCP server messages to fly around your ISP's local subnet if someone runs the server by accident. For this reason, it's especially important to filter DHCP messages to limit traffic between your client and your specific ISP DHCP server as much as possible.

Table 3.18 lists the DHCP message type descriptions as quoted from RFC 2131, "Dynamic Host Configuration Protocol."

**Table 3.18 DHCP Message Types**

| DHCP Message | Description |
|---|---|
| `DHCPDISCOVER` | Client broadcast to locate available servers. |
| `DHCPOFFER` | Server to client in response to `DHCPDISCOVER` with offer of configuration parameters. |
| `DHCPREQUEST` | Client message to servers either (a) requesting offered parameters from one server and implicitly declining offers from all others; (b) confirming correctness of previously allocated address after, e.g., system reboot; or (c) extending the lease on a particular network address. |
| `DHCPACK` | Server to client with configuration parameters, including committed network address. |
| `DHCPNAK` | Server to client indicating client's notion of network address is incorrect (e.g., client has moved to new subnet) or client's lease has expired. |
| `DHCPDECLINE` | Client to server indicating network address is already in use. |
| `DHCPRELEASE` | Client to server relinquishing network address and canceling remaining lease. |
| `DHCPINFORM` | Client to server, asking only for local configuration parameters; client already has externally configured address. (Not supported in Red Hat 6.0.) |

In essence, when the DHCP client initializes, it broadcasts a `DHCPDISCOVER` query to discover whether any DHCP servers are available. Any servers receiving the query may respond with a `DHCPOFFER` message indicating their willingness to function as server to this client, and include the configuration parameters they have to offer. The client broadcasts a `DHCPREQUEST` message to both accept one of the servers and inform any remaining servers that it has chosen to decline their offers. The chosen server responds with a `DHCPACK` message, indicating confirmation of the parameters it originally offered. Address assignment is complete at this point. Periodically, the client will send a `DHCPREQUEST` message requesting a renewal on the IP address lease. If the lease is renewed, the server responds with a `DHCPACK` message. Otherwise, the client falls back to the initialization process. Table 3.19 lists the local client to remote server exchange protocol for the DHCP service.

The DHCP protocol is far more complicated than this brief summary, but the summary describes the essentials of the typical client and server exchange.

**Table 3.19 DHCP Protocol**

| Description | Protocol | Remote Address | Remote Port | In/ Out | Local Address | Local Port |
|---|---|---|---|---|---|---|
| DHCPDISCOVER; DHCPREQUEST | UDP | 255.255.255.255 | 67 | Out | 0.0.0.0 | 68 |
| DHCPOFFER | UDP | 0.0.0.0 | 67 | In | 255.255.255.255 | 68 |
| DHCPOFFER | UDP | DHCP SERVER | 67 | In | 255.255.255.255 | 68 |
| DHCPREQUEST; DHCPDECLINE | UDP | DHCP SERVER | 67 | Out | 0.0.0.0 | 68 |
| DHCPACK; DHCPNAK | UDP | DHCP SERVER | 67 | In | ISP/netmask | 68 |
| DHCPACK | UDP | DHCP SERVER | 67 | In | IPADDR | 68 |
| DHCPREQUEST; DHCPRELEASE | UDP | DHCP SERVER | 67 | Out | IPADDR | 68 |

The following firewall rules allow communication between your DHCP client and a remote server:

```
DHCP_SERVER="my.dhcp.server"            # if you use one

# INIT or REBINDING: No lease or Lease time expired.

ipchains -A output -i $EXTERNAL_INTERFACE -p udp \
         -s $BROADCAST_0 68 \
         -d $BROADCAST_1 67 -j ACCEPT
```

```
# Getting renumbered

ipchains -A input  -i $EXTERNAL_INTERFACE -p udp \
         -s $BROADCAST_0 67 \
         -d $BROADCAST_1 68 -j ACCEPT

ipchains -A input  -i $EXTERNAL_INTERFACE -p udp \
         -s $DHCP_SERVER 67 \
         -d $BROADCAST_1 68 -j ACCEPT

ipchains -A output -i $EXTERNAL_INTERFACE -p udp \
         -s $BROADCAST_0 68 \
         -d $DHCP_SERVER 67 -j ACCEPT

# As a result of the above, we're supposed to change our IP
# address with this message, which is addressed to our new
# address before the dhcp client has received the update.

ipchains -A input  -i $EXTERNAL_INTERFACE -p udp \
         -s $DHCP_SERVER 67 \
         -d $MY_ISP 68 -j ACCEPT

ipchains -A input  -i $EXTERNAL_INTERFACE -p udp \
         -s $DHCP_SERVER 67 \
         -d $IPADDR 68 -j ACCEPT

ipchains -A output -i $EXTERNAL_INTERFACE -p udp \
         -s $IPADDR 68 \
         -d $DHCP_SERVER 67 -j ACCEPT
```

Notice that DHCP traffic cannot be completely limited to your DHCP server. During initialization sequences, when your client doesn't yet have an assigned IP address or even the server's IP address, packets are broadcast rather than sent point-to-point.

## Accessing Remote Network Time Servers (UDP 123)

Network time services such as NTP allow access to one or more public Internet time providers. This is useful to maintain an accurate system clock, particularly if your internal clock tends to drift, and to establish the correct time and date at bootup or after a power loss. A small system user should use the service only as a client. Few, if any, small sites have a satellite link to Greenwich, England, a radio link to the United States atomic clock, or an atomic clock of their own lying around.

`xntpd` is the server daemon. In addition to providing time service to clients, `xntpd` also uses a peer-to-peer relationship among servers. Few small sites require the extra precision `xntpd` provides. `ntpdate` is the client program, and can use either client-to-server or peer-to-peer communication. The client program is all a small site will need. Table 3.20 lists only the client/server exchange protocol for the NTP service.

**Table 3.20 NTP Protocol**

| Description | Protocol | Remote Address | Remote Port | In/ Out | Local Address | Local Port |
|---|---|---|---|---|---|---|
| Local client query | UDP | timeserver | 123 | Out | IPADDR | 1024:65535 |
| Remote server response | UDP | timeserver | 123 | In | IPADDR | 1024:65535 |

As a client, you can use `ntpdate` to periodically query a series of public time service providers from a `cron` job. These hosts would be individually specified in a series of firewall rules:

```
ipchains -A output -i $EXTERNAL_INTERFACE -p udp \
         -s $IPADDR $UNPRIVPORTS \
         -d <my.time.provider> 123 -j ACCEPT

ipchains -A input  -i $EXTERNAL_INTERFACE -p udp \
         -s <my.time.provider> 123 \
         -d $IPADDR $UNPRIVPORTS -j ACCEPT
```

# Logging Denied Incoming Packets

Any packet matching a rule can be logged by adding the `-l` option to its `ipchains` rule. Some of the rules presented previously had logging enabled. The IP address spoofing rules are examples.

Rules can be defined for the explicit purpose of logging certain kinds of packets. Most typically, packets of interest are suspicious packets indicating some sort of probe or scan. Because all packets are denied by default, if logging is desired for certain packet types, explicit rules must be defined before the packet falls off the end of the chain and the default policy takes effect. Essentially, out of all the denied packets, you might be interested in logging some of them.

Which packets are logged is an individual matter. Some people want to log all denied packets. For other people, logging all denied packets could soon overflow their system logs. Some people, secure in the knowledge that the packets are denied, don't care about them and don't want to know about them. Other people are interested in the obvious port scans or some particular packet type.

Because of the first-matching-rule-wins behavior, you could log all denied incoming packets with a single rule:

```
ipchains -A input -i $EXTERNAL_INTERFACE -j DENY -l
```

For some people, this will produce too many log entries—or too many uninteresting log entries. For example, you might want to log all denied incoming ICMP traffic with the exception of `ping`, because it is a common service, regardless of whether your site responds to `ping` requests:

```
ipchains -A input -i $EXTERNAL_INTERFACE -p icmp \
         -s $ANYWHERE 1:7 -d $IPADDR -j DENY -l

ipchains -A input -i $EXTERNAL_INTERFACE -p icmp \
         -s $ANYWHERE 9:18 -d $IPADDR -j DENY -l
```

You might want to log denied incoming TCP traffic to all ports, and denied incoming UDP traffic to your privileged ports:

```
ipchains -A input -i $EXTERNAL_INTERFACE -p tcp \
         -d $IPADDR -j DENY -l

ipchains -A input -i $EXTERNAL_INTERFACE -p udp \
         -d $IPADDR $PRIVPORTS -j DENY -l
```

Then again, you might want to log all denied privileged port access, with the exception of commonly probed ports you don't offer service on anyway:

```
ipchains -A input -i $EXTERNAL_INTERFACE -p tcp \
         -d $IPADDR 0:19 -j DENY -l

# skip ftp, telnet, ssh
ipchains -A input -i $EXTERNAL_INTERFACE -p tcp \
         -d $IPADDR 24 -j DENY -l

# skip smtp
ipchains -A input -i $EXTERNAL_INTERFACE -p tcp \
         -d $IPADDR 26:78 -j DENY -l

# skip finger, www
ipchains -A input -i $EXTERNAL_INTERFACE -p tcp \
         -d $IPADDR 81:109 -j DENY -l

# skip pop-3, sunrpc
ipchains -A input -i $EXTERNAL_INTERFACE -p tcp \
         -d $IPADDR 112:136 -j DENY -l

# skip NetBIOS
ipchains -A input -i $EXTERNAL_INTERFACE -p tcp \
         -d $IPADDR 140:142 -j DENY -l

# skip imap
ipchains -A input -i $EXTERNAL_INTERFACE -p tcp \
         -d $IPADDR 144:442 -j DENY -l

# skip secure_web/SSL
ipchains -A input -i $EXTERNAL_INTERFACE -p tcp \
         -d $IPADDR 444:65535 -j DENY -l
```

```
#UDP rules
ipchains -A input -i $EXTERNAL_INTERFACE -p udp \
         -d $IPADDR 0:110 -j DENY -l

# skip sunrpc
ipchains -A input -i $EXTERNAL_INTERFACE -p udp \
         -d $IPADDR 112:160 -j DENY -l

# skip snmp
ipchains -A input -i $EXTERNAL_INTERFACE -p udp \
         -d $IPADDR 163:634 -j DENY -l

# skip NFS mountd
ipchains -A input -i $EXTERNAL_INTERFACE -p udp \
         -d $IPADDR 636:5631 -j DENY -l

# skip pcAnywhere
ipchains -A input -i $EXTERNAL_INTERFACE -p udp \
         -d $IPADDR 5633:31336 -j DENY -l

# skip BackOrifice
ipchains -A input -i $EXTERNAL_INTERFACE -p udp \
         -d $IPADDR 31338:33433 -j DENY -l

# skip traceroute
ipchains -A input -i $EXTERNAL_INTERFACE -p udp \
         -s $ANYWHERE 32679:65535 \
         -d $IPADDR 33434:33523 -j DENY -l

# skip the rest
ipchains -A input -i $EXTERNAL_INTERFACE -p udp \
         -d $IPADDR 33434:65535 -j DENY -l
```

## Denying Access to Problem Sites Up Front

If some site is making a habit of scanning your machine or otherwise being a nuisance, you might decide to deny it access to everything, at least until the problem behavior is corrected.

One way to do this without editing the `rc.firewall` script each time is to include a separate file of specific denial rules. By inserting the rules into the `input` chain rather than appending them, the site will be blocked even if subsequent rules would otherwise allow them access to some service. The file is named `/etc/rc.d/rc.firewall.blocked`. To avoid a possible runtime error, you check for the file's existence before trying to include it:

```
# Refuse packets claiming to be from the banned list
if [ -f /etc/rc.d/rc.firewall.blocked ]; then
     . /etc/rc.d/rc.firewall.blocked
fi
```

An example of a global denial rule in the `rc.firewall.blocked` file could be:

```
ipchains -I input -i $EXTERNAL_INTERFACE -s <address/mask> -j DENY
```

Any packet from this source address range is denied, regardless of message protocol type or source or destination port.

At this point, the firewall rules are defined. When the firewall rules are installed in the kernel as a functional firewall, you can connect your Linux machine to the Internet with a good measure of confidence that your system is secure against most outside attacks.

# Enabling LAN Access

If the firewall machine sits between the Internet and a LAN, machines on the LAN have access neither to the firewall machine's internal network interface nor to the Internet. Chapter 4 covers LAN firewall issues in depth. A small site, particularly a home site, won't need or have the resources to implement the firewall architecture presented in Chapter 4. For the average home site, and for many small business sites as well, the single-machine firewall developed in this chapter is sufficient.

To support a LAN behind the firewall, a few more rules are needed to enable access to the firewall machine's internal network interface and to pass internal traffic through to the Internet. When the firewall machine serves in this capacity, with two or more network interfaces, it's called a bastion firewall, or a screened-subnet firewall.

## Enabling LAN Access to the Firewall's Internal Network Interface

For a home or small business setup, there is probably little reason to limit direct access to the firewall machine from the internal LAN. This rule pair allows open communication between the firewall machine and the LAN:

```
LAN_INTERFACE_1="eth1"              # internal LAN interface

LAN_1="192.168.1.0/24"              # your (private) LAN address range
LAN__IPADDR_1="192.168.1.1"         # your internal interface address

ipchains -A input  -i $LAN_INTERFACE_1 \
         -s $LAN_1 -j ACCEPT

ipchains -A output -i $LAN_INTERFACE_1 \
         -d $LAN_1 -j ACCEPT
```

Notice that this rule pair allows LAN access to the firewall machine. The LAN does not yet have Internet access through the firewall. Because a firewall machine, by definition, does not route traffic dynamically, or automatically using static routes (unless the machine is misconfigured), additional firewall rules are necessary to route local traffic onward.

## Enabling LAN Access to the Internet: IP Forwarding and Masquerading

At this point, selected ports are open for either client or server communication, or both, between remote machines and the firewall machine's external network interface. Local communication between the LAN and firewall machine is completely open through the firewall's internal network interface. Internal machines on the LAN do not yet have access to the Internet, however. Allowing Internet access is a two-step process. Communication between the LAN and the Internet must be both forwarded and masqueraded.

IP forwarding is a kernel service allowing the Linux machine to act as a router between two networks, forwarding traffic from one network to the other. With a LAN, IP forwarding must be enabled in the routing section of the network configuration. With a deny-everything-by-default firewall policy in force, however, forwarded packets can't cross between the two interfaces until specific rules allow it.

Few home-based systems should or will want to forward internal traffic directly. IP addresses taken from the Class A, B, or C private address ranges require IP masquerading (another kernel service), or application-level proxying to substitute the private LAN IP address with the public IP address of the firewall machine's external interface. Packets with private source addresses should not cross beyond the firewall machine out to the Internet, and if they do, they might not be routed to their destination indefinitely. Even if your site has registered, static IP addresses, IP masquerading and application-level proxy servers are two of the best ways to secure and transparently isolate your internal machines from the Internet.

On the `ipchains` administration level, forwarding and masquerading appear to be different aspects of the same service. (In fact, they are separate mechanisms. But the user interfaces to the two services are combined in the firewall administration program.) Forwarding routes LAN traffic from the firewall's internal interface out through the external interface to the Internet. Before the packets are placed on the firewall machine's external interface output queue, the masquerading service replaces the packet's source address with that of the firewall machine's external interface's public IP address. Forwarding and masquerading together let the firewall machine act as a filtering, proxying router.

The following rule demonstrates how to forward and masquerade all internal traffic out through the external interface. The `ACCEPT` and `DENY` rules for the external interface's `output` chain are applied after the forwarding rules are applied, so even though everything is allowed to be forwarded and masqueraded between the two network interfaces, only those packets allowed out by the firewall rules for the external interface will actually pass through:

```
ipchains -A forward -i $EXTERNAL_INTERFACE -s $LAN_1 -j MASQ
```

Masquerading rules can take source and destination address and port arguments, just as the other rules do. The network interface argument is the name of the forwarding (external) interface, not the packets' local network interface. Although the rules in the example allow everything, you could just as easily define specific rules to masquerade and forward only specific services, only TCP traffic, and so forth.

# Installing the Firewall

As a shell script, installation is simple. The script should be owned by `root`:

```
chown root.root /etc/rc.d/rc.firewall
```

The script should be writable and executable by `root` alone. Ideally, the general user should not have read access:

```
chmod ug=rwx /etc/rc.d/rc.firewall
```

To initialize or reinitialize the firewall at any time, execute the script from the command line. There is no need to reboot:

```
sh /etc/rc.d/rc.firewall
```

How the script is executed at boot time varies based on whether you have a registered, static IP address or a dynamic, DHCP-assigned IP address.

## Installing a Firewall with a Static IP Address

If you have a static IP address, the simplest way to initialize the firewall is to edit `/etc/rc.d/rc.local` and add the following line to the end of the file:

```
sh /etc/rc.d/rc.firewall
```

If hostnames are used in the firewall rules, the important thing to remember is that DNS traffic must be enabled before the hostnames are encountered in the script. If a local name server is configured, the system automatically starts `named` before running `rc.local` at boot time or later if changing runlevels.

## Installing a Firewall with a Dynamic IP Address

If you have a dynamic IP address, you will discover that firewall installation support was (annoyingly) discontinued as of Red Hat 6.0. With luck, enough irate DHCP clients will complain to get firewall support reinstated. In the meantime, you have to reconfigure your system's default installation setup for DHCP. The following steps work for earlier versions of `/sbin/dhcpcd` prior to Red Hat 6.0. If you've upgraded from an earlier release, these steps reinstate your prior environment:

1. Red Hat 6.0 replaced the DHCP client, `dhcpcd`, with a new client, `pump`. `pump` doesn't provide a mechanism for executing a script when your IP address is assigned or reassigned. Consequently, without editing one of the network startup

scripts, /sbin/ifup, you have no means of storing the dynamic information the DHCP server has provided, nor do you have a way of automatically restarting the firewall script if your IP address is reassigned after a lease revocation. You have to edit the executable script, /sbin/ifup, to use the older /sbin/dhcpcd instead of /sbin/pump.

Refer to the section "dhcpcd Support in /sbin/ifup" in Appendix B for examples of code.

2. Create a new executable shell script, /etc/sysconfig/network-scripts/ifdhcpc-done. This file was provided as part of the Red Hat distribution until release 6.0. It was run by dhcpcd after IP address assignment or reassignment. pump doesn't support running a script.

   ifdhcpc-done's original, primary purpose was to provide a mechanism to inform /sbin/ifup as to whether dhcpcd succeeded in getting its dynamic information from the DHCP server or not. Depending on the particular Red Hat release, the file also performed a few other file updates.

   ifdhcpc-done is the perfect place to execute /etc/rc.d/rc.firewall from because ifdhcpc-done is executed each time the IP address is assigned or changed. It's also a useful place from which to perform several other functions. Among them are setting the system's domain name, updating /etc/hosts with the current IP address, updating /etc/resolv.conf if you run your own name server, updating /etc/named.conf if you run your own name server and forward queries to your ISP's name servers, and providing a mechanism for relaying the current IP address and name server addresses to the firewall script.

   Refer to the section "Updating Dynamic Addresses and Installing the Firewall from /etc/sysconfig/network-scripts/ifdhcpc-done" in Appendix B for examples of code.

3. Create dhcpcd's configuration directory, /etc/dhcpc:

   ```
   mkdir /etc/dhcpc
   ```

   pump doesn't use this directory. dhcpcd expects the directory to exist.

4. The firewall script itself needs to include IPADDR and the NAMESERVERs from /etc/dhcpc/dhcpd-eth0.info. These addresses are provided by the DHCP server. Name server addresses are quite stable. Your IP address can change relatively frequently, depending on your ISP's DHCP server configuration.

## Summary

This chapter leads you through the processes involved in developing a standalone firewall using `ipchains`. The deny-by-default policy was established. Initial potential problems were addressed, such as source address spoofing and protecting services running on unprivileged ports. ICMP messages, the control and status messages used by the underlying IP network layer, were handled. DNS name service, which underlies all network services, and `AUTH` user identification service, which supports several common network services, were handled. Examples of rules for popular network services were shown. Examples of controlling the level of logging produced were demonstrated. Finally, the issues involved in firewall installation were described, both for sites with a static IP address and sites with a dynamically assigned IP address.

At the end, the firewall script was very slightly extended to add support to serve as a bastion firewall for a small LAN. Complete script examples for both `ipchains` and `ipfwadm` are included in Appendix B.

Chapter 4 uses the bastion firewall as the basis for building a more complicated firewall architecture. A screened subnet architecture using two firewalls separating a DMZ perimeter network is described in Chapter 4. A small business could easily have the need and the resources for this more elaborate configuration.