Linux IPCHAINS−HOWTO

Table of Contents

Table of Contents

Table of Contents

Linux IPCHAINS−HOWTO

Rusty Russell

v1.0.8, Tue Jul 4 14:20:53 EST 2000

This document aims to describe how to obtain, install and configure the enhanced IP firewalling chains software for Linux, and some ideas on how you might use them.

1.Introduction

2.Packet Filtering Basics

3.I'm confused! Routing, masquerading, portforwarding, ipautofw...

4.IP Firewalling Chains

5.Miscellaneous.

6.Common Problems

7.A Serious Example.

8.Appendix: Differences between ipchains and ipfwadm.

9.Appendix: Using the ipfwadm−wrapper script.

10.Appendix: Thanks.

1.Introduction

This is the Linux IPCHAINS−HOWTO; see Where? for the master site, which contains the latest copy. You should read the Linux NET−3−HOWTO as well. The IP−Masquerading HOWTO, the PPP−HOWTO, the Ethernet−HOWTO and the Firewall HOWTO might make interesting reading. (Then again, so might the alt.fan.bigfoot FAQ).

If packet filtering is passe to you, read Section Why?, Section How?, and scan through the titles in Section IP Firewalling Chains.

If you are converting from `ipfwadm`, read Section Introduction, Section How?, and Appendices in section Differences between ipchains and ipfwadm and section Using the `ipfwadm−wrapper' script.

1.1 What?

Linux `ipchains` is a rewrite of the Linux IPv4 firewalling code (which was mainly stolen from BSD) and a rewrite of `ipfwadm`, which was a rewrite of BSD's `ipfw`, I believe. It is required to administer the IP packet filters in Linux kernel versions 2.1.102 and above.

1.2 Why?

The older Linux firewalling code doesn't deal with fragments, has 32−bit counters (on Intel at least), doesn't allow specification of protocols other than TCP, UDP or ICMP, can't make large changes atomically, can't specify inverse rules, has some quirks, and can be tough to manage (making it prone to user error).

1.3 How?

Currently the code is in the mainstream kernel from 2.1.102. For the 2.0 kernel series, you will need to download a kernel patch from the web page. If your 2.0 kernel is more recent than the supplied patch, the older patch should be OK; this part of the 2.0 kernels is fairly stable (eg. the 2.0.34 kernel patch works just fine on the 2.0.35 kernel). Since the 2.0 patch is incompatible with the ipportfw and ipautofw patches, I don't recommend applying it unless you really need some functionality that ipchains offers.

1.4 Where?

The official page is in three places: Thanks to Penguin ComputingThanks to the SAMBA TeamThanks to Jim Pick

There is a mailing list for bug reports, discussion, development and usage. Join the mailing list by sending a message containing the word ``subscribe ipchains−list'' to subscribe at east.balius.com. To mail to everyone on the list use ipchains−list at east.balius.com.

2.Packet Filtering Basics

2.1 What?

All traffic through a network is sent in the form of **packets**. For example, downloading this package (say it's 50k long) might cause you to receive 36 or so packets of 1460 bytes each, (to pull numbers at random).

The start of each packet says where it's going, where it came from, the type of the packet, and other administrative details. This start of the packet is called the **header**. The rest of the packet, containing the actual data being transmitted, is usually called the **body**.

Some protocols, such **TCP**, which is used for web traffic, mail, and remote logins, use the concept of a `connection' −− before any packets with actual data are sent, various setup packets (with special headers) are exchanged saying `I want to connect', `OK' and `Thanks'. Then normal packets are exchanged.

A packet filter is a piece of software which looks at the *header* of packets as they pass through, and decides the fate of the entire packet. It might decide to **deny** the packet (ie. discard the packet as if it had never received it), **accept** the packet (ie. let the packet go through), or **reject** the packet (like deny, but tell the source of the packet that it has done so).

Under Linux, packet filtering is built into the kernel, and there are a few trickier things we can do with packets, but the general principle of looking at the headers and deciding the fate of the packet is still there.

2.2 Why?

Control. Security. Watchfulness.

Control:

when you are using a Linux box to connect your internal network to another network (say, the Internet) you have an opportunity to allow certain types of traffic, and disallow others. For example, the header of a packet contains the destination address of the packet, so you can prevent packets going to a certain part of the outside network. As another example, I use Netscape to access the Dilbert archives. There are advertisements from doubleclick.net on the page, and Netscape wastes my time by cheerfully downloading them. Telling the packet filter not to allow any packets to or from the addresses owned by doubleclick.net solves that problem (there are better ways of doing this though).

Security:

when your Linux box is the only thing between the chaos of the Internet and your nice, orderly network, it's nice to know you can restrict what comes tromping in your door. For example, you might allow anything to go out from your network, but you might be worried about the well−known `Ping of Death' coming in from malicious outsiders. As another example, you might not want outsiders telnetting to your Linux box, even though all your accounts have passwords; maybe you want (like most people) to be an observer on the Internet, and not a server (willing or otherwise) −− simply don't let anyone connect in, by having the packet filter reject incoming packets used to set up connections.

Watchfulness:

sometimes a badly configured machine on the local network will decide to spew packets to the outside world. It's nice to tell the packet filter to let you know if anything abnormal occurs; maybe you can do something about it, or maybe you're just curious by nature.

2.3 How?

A Kernel With Packet Filtering

You need a kernel which has the new IP firewall chains in it. You can tell if the kernel you are running right now has this installed by looking for the file `/proc/net/ip_fwchains'. If it exists, you're in.

If not, you need to make a kernel that has IP firewall chains. First, download the source to the kernel you want. If you have a kernel numbered 2.1.102 or higher, you won't need to patch it (it's in the mainstream kernel now). Otherwise, apply the patch from the web page listed above, and set the configuration as detailed below. If you don't know how to do this, don't panic −− read the Kernel−HOWTO.

The configuration options you will need to set *for the 2.0−series kernel* are:

```
        CONFIG_EXPERIMENTAL=y
```

```
CONFIG_FIREWALL=y
CONFIG_IP_FIREWALL=y
CONFIG_IP_FIREWALL_CHAINS=y
```

For the *2.1 or 2.2 series kernels*:

```
CONFIG_FIREWALL=y
CONFIG_IP_FIREWALL=y
```

The tool `ipchains` talks to the kernel and tells it what packets to filter. Unless you are a programmer, or overly curious, this is how you will control the packet filtering.

ipchains

The `ipchains` tool inserts and deletes rules from the kernel's packet filtering section. This means that whatever you set up, it will be lost upon reboot; see Making Rules Permanent for how to make sure they are restored the next time Linux is booted.

`ipchains` replaces `ipfwadm`, which was used for the old IP Firewall code. There is a set of useful scripts available from the ipchains ftp site:

http://netfilter.filewatcher.org/ipchains/ipchains−scripts−1.1.2.tar.gz

This contains a shell script called `ipfwadm−wrapper` which allows you to do packet filtering as it was done before. You probably shouldn't use this script unless you want a quick way of upgrading a system which uses `ipfwadm` (it's slower, and doesn't check arguments, etc). In that case, you don't need this HOWTO much either.

See Appendix Differences between ipchains and ipfwadm and Appendix Using the `ipfwadm−wrapper' script for more details on `ipfwadm` issues.

Making Rules Permanent

Your current firewall setup is stored in the kernel, and thus will be lost on reboot. I recommend using the `ipchains−save' and `ipchains−restore' scripts to make your rules permanent. To do this, set up your rules, then run (as root):

```
# ipchains-save > /etc/ipchains.rules
#
```

Create a script like the following:

```
#! /bin/sh
# Script to control packet filtering.

# If no rules, do nothing.
[ -f /etc/ipchains.rules ] || exit 0

case "$1" in
    start)
        echo -n "Turning on packet filtering:"
        /sbin/ipchains-restore < /etc/ipchains.rules || exit 1
        echo 1 > /proc/sys/net/ipv4/ip_forward
        echo "."
        ;;
    stop)
        echo -n "Turning off packet filtering:"
        echo 0 > /proc/sys/net/ipv4/ip_forward
        /sbin/ipchains -F
        /sbin/ipchains -X
        /sbin/ipchains -P input ACCEPT
        /sbin/ipchains -P output ACCEPT
        /sbin/ipchains -P forward ACCEPT
        echo "."
        ;;
    *)
        echo "Usage: /etc/init.d/packetfilter {start|stop}"
        exit 1
        ;;
esac

exit 0
```

Make sure this is run early in the bootup procedure. In my case (Debian 2.1), I make a symbolic link called `S39packetfilter' in the `/etc/rcS.d' directory (this will be run before S40network).

3.I'm confused! Routing, masquerading, portforwarding, ipautofw...

This HOWTO is about packet filtering. This means deciding whether a packet should be allowed to pass or not. However, Linux being the hacker's playground that it is, you probably want to do more than that.

One problem is that the same tool (``ipchains'') is used to control both masquerading and transparent proxying, although these are notionally separate from packet filtering (the current Linux implementation blurs these together unnaturally, leaving the impression that they are closely related).

Masquerading and proxying are covered by separate HOWTOs, and the auto forwarding and port forwarding features are controlled by separate tools, but since so many people keep asking me about it, I'll include a set of common scenarios and indicate when each one should be applied. The security merits of each setup will not be discussed here.

3.1 Rusty's Three−Line Guide To Masquerading

This assumes that your **external** interface is called `ppp0'. Use ifconfig to find out, and adjust to taste.

```
# ipchains −P forward DENY
# ipchains −A forward −i ppp0 −j MASQ
# echo 1 > /proc/sys/net/ipv4/ip_forward
```

3.2 Gratuitous Promotion: WatchGuard Rules

You can buy off−the−shelf firewalls. An excellent one is WatchGuard's FireBox. It's excellent because I like it, it's secure, it's Linux−based, and because they funded the maintenance of ipchains as well as the new firewalling code (for 2.4). In short, WatchGuard were paying for me to eat while I work for you. So please consider their stuff.

http://www.watchguard.com

3.3 Common Firewall−like Setups

You run littlecorp.com. You have an internal network, and a single dialup (PPP) connection to the Internet (firewall.littlecorp.com which is 1.2.3.4). You run Ethernet on your local network, and your personal machine is called "myhost".

This section will illustrate the different arrangement which are common. Read carefully, because they are each subtly different.

Private Network: Traditional Proxies

In this scenario, packets from the private network never traverse the Internet, and vice versa. The IP addresses of the private network should be assigned from the RFC1918 Address Allocation for Private Internets (ie. 10.*.*.*, 172.16.*.*−172.31.*.* or 192.168.*.*).

The only way things ever connect to the Internet is by connecting to the firewall, which is the only machine on both networks which connects onwards. You run a program (on the firewall) called a proxy to do this (there are proxies for FTP, web access, telnet, RealAudio, Usenet News and other services). See the Firewall HOWTO.

Any services you wish the Internet to access must be on the firewall. (But see Limited Internal Services below).

Example: Allowing web access from private network to the Internet.

1. The private network is assigned 192.168.1.* addresses, with myhost being 192.168.1.100, and the firewall's Ethernet interface being assigned 192.168.1.1.
2. A web proxy (eg. "squid") is installed and configured on the firewall, say running on port 8080.
3. Netscape on the private network is configured to use the firewall port 8080 as a proxy.
4. DNS does not need to be configured on the private network.
5. DNS does need to be configured on the firewall.
6. No default route (aka gateway) needs to be configured on the private network.

Netscape on myhost reads http://slashdot.org.

1. Netscape connects to the firewall port 8080, using port 1050 on myhost. It asks for the web page of "http://slashdot.org".
2. The proxy looks up the name "slashdot.org", and gets 207.218.152.131. It then opens a connection to that IP address (using port 1025 on the firewall's external interface), and asks the web server (port 80) for the web page.
3. As it receives the web page from its connection to the web server, it copies the data to the connection from Netscape.
4. Netscape renders the page.

ie. From slashdot.org's point of view, the connection is made from 1.2.3.4 (firewall's PPP interface) port 1025 to 207.218.152.131 (slashdot.org) port 80. From myhost's point of view, the connection is made from 192.168.1.100 (myhost) port 1050, to 192.168.1.1 (firewall's Ethernet interface) port 8080.

Private Network: Transparent Proxies

In this scenario, packets from the private network never traverse the Internet, and vice versa. The IP addresses of the private network should be assigned from the RFC1918 Address Allocation for Private Internets (ie. 10.*.*.*, 172.16.*.*−172.31.*.* or 192.168.*.*).

The only way things ever connect to the Internet is by connecting to the firewall, which is the only machine on both networks, which connects onwards. You run a program (on the firewall) called a transparent proxy to do this; the kernel sends outgoing packets to the transparent proxy instead of sending them onwards (ie. it bastardizes routing).

Transparent proxying means that the clients don't need to know there is a proxy involved.

Any services you wish the Internet to access must be on the firewall. (But see Limited Internal Services below).

Example: Allowing web access from private network to the Internet.

1. The private network is assigned 192.168.1.* addresses, with myhost being 192.168.1.100, and the firewall's Ethernet interface being assigned 192.168.1.1.
2. A transparent web proxy (I believe there are patches for squid to allow it to operate in this manner, or try "transproxy") is installed and configured on the firewall, say running on port 8080.
3. The kernel is told to redirect connections to port 80 to the proxy, using ipchains.
4. Netscape on the private network is configured to connect directly.
5. DNS needs to be configured on the private network (ie. you need to run a DNS server as a proxy on the firewall).
6. The default route (aka gateway) needs to be configured on the private network, to send packets to the firewall.

Netscape on myhost reads http://slashdot.org.

1. Netscape looks up the name "slashdot.org", and gets 207.218.152.131. It then opens a connection to that IP address, using local port 1050, and asks the web server (port 80) for the web page.
2. As the packets from myhost (port 1050) to slashdot.org (port 80) pass through the firewall, they are redirected to the waiting transparent proxy on port 8080. The transparent proxy opens a connection (using local port 1025) to 207.218.152.131 port 80 (which is where the original packets were going).
3. As the proxy receives the web page from its connection to the web server, it copies the data to the connection from Netscape.
4. Netscape renders the page.

ie. From slashdot.org's point of view, the connection is made from 1.2.3.4 (firewall's PPP interface) port 1025 to 207.218.152.131 (slashdot.org) port 80. From myhost's point of view, the connection is made from 192.168.1.100 (myhost) port 1050, to 207.218.152.131 (slashdot.org) port 80, but it's actually talking to the transparent proxy.

Private Network: Masquerading

In this scenario, packets from the private network never traverse the Internet without special treatment, and vice versa. The IP addresses of the private network should be assigned from the RFC1918 Address Allocation for Private Internets (ie. 10.*.*.*, 172.16.*.*−172.31.*.* or 192.168.*.*).

Instead of using a proxy, we use a special kernel facility called "masquerading". Masquerading rewrites packets as they pass through the firewall, so that they always seem to come from the firewall itself. It then rewrites the responses so that they look like they are going to the original recipient.

Masquerading has separate modules to handle "tricky" protocols, such as FTP, RealAudio, Quake, etc. For really hard−to−handle protocols, the "auto forwarding" facility can handle some of them by automatically setting up port forwarding for related sets of ports: look for ``ipportfw'' (2.0 kernels) or ``ipmasqadm'' (2.1 kernels).

Any services you wish the Internet to access must be on the firewall. (But see Limited Internal Services below).

Example: Allowing web access from private network to the Internet.

1. The private network is assigned 192.168.1.* addresses, with myhost being 192.168.1.100, and the firewall's Ethernet interface being assigned 192.168.1.1.
2. The firewall is set up to masquerade any packets coming from the private network and going to port 80 on an Internet host.
3. Netscape is configured to connect directly.
4. DNS must be configured correctly on the private network.
5. The firewall should be the default route (aka gateway) for the private network.

Netscape on myhost reads http://slashdot.org.

1. Netscape looks up the name "slashdot.org", and gets 207.218.152.131. It then opens a connection to that IP address, using local port 1050, and asks the web server (port 80) for the web page.
2. As the packets from myhost (port 1050) to slashdot.org (port 80) pass through the firewall, they are rewritten to come from the PPP interface of the firewall, port 65000. The firewall has a valid Internet address (1.2.3.4) so reply packets from slashdot.org get routed back OK.
3. As packets from slashdot.org (port 80) to firewall.littlecorp.com (port 65000) come in, they are rewritten to go to myhost, port 1050. This is the real magic of masquerading: it remembers when it rewrites outgoing packets to it can write them back as replies come in.
4. Netscape renders the page.

ie. From the slashdot.org's point of view, the connection is made from 1.2.3.4 (firewall's PPP interface) port 65000 to 207.218.152.131 (slashdot.org) port 80. From the myhost's point of view, the connection is made from 192.168.1.100 (myhost) port 1050, to 207.218.152.131 (slashdot.org) port 80.

Public Network

In this scenario, your personal network is a part of the Internet: packets can flow without change across both networks. The IP addresses of the internal network must be assigned by applying for a block of IP addresses, so the rest of the network will know how to get packets to you. This implies a permanent connection.

In this role, packet filtering is used to restrict which packets can be forwarded between your network and the rest of the Internet, eg. to restrict the rest of the Internet to only accessing your internal web servers.

Example: Allowing web access from private network to the Internet.

1. Your internal network is assigned according to the IP address block you have registered, (say 1.2.3.*).
2. The firewall is set up to allow all traffic.
3. Netscape is configured to connect directly.
4. DNS must be configured correctly on your network.
5. The firewall should be the default route (aka gateway) for the private network.

Netscape on myhost reads http://slashdot.org.

1. Netscape looks up the name "slashdot.org", and gets 207.218.152.131. It then opens a connection to that IP address, using local port 1050, and asks the web server (port 80) for the web page.
2. Packets pass through your firewall, just as they pass through several other routers between you and slashdot.org.
3. Netscape renders the page.

ie. There is only one connection: from 1.2.3.100 (myhost) port 1050, to 207.218.152.131 (slashdot.org) port 80.

Limited Internal Services

There are a few tricks you can pull to allow the Internet to access your internal services, rather than running the services on the firewall. These will work with either a proxy or masquerading based approach for external connections.

The simplest approach is to run a "redirector", which is a poor−man's proxy which waits for a connection on a given port, and then open a connection a fixed internal host and port, and copies data between the two connections. An example of this is the "redir" program. From the Internet point of view, the connection is made to your firewall. From your internal server's point of view, the connection is made from the internal interface of the firewall to the server.

Another approach (which requires a 2.0 kernel patched for ipportfw, or a 2.1 or later kernel) is to use port forwarding in the kernel. This does the same job as "redir" in a different way: the kernel rewrites packets as they pass through, changing their destination address and ports to point them at an internal host and port. From the Internet's point of view, the connection is made to your firewall. From your internal server's point of view, a direct connection is made from the Internet host to the server.

3.4 More Information on Masquerading

David Ranch has written an excellent new HOWTO on Masquerading, which has a large amount of overlap with this HOWTO. You can currently find that HOWTO at

http://www.linuxdoc.org/HOWTO/IP−Masquerade−HOWTO.html

The official Masquerading home page is at

http://ipmasq.cjb.net

4.IP Firewalling Chains

This section describes all you really need to know to build a packet filter that meets your needs.

4.1 How Packets Traverse The Filters

The kernel starts with three lists of rules; these lists are called **firewall chains** or just **chains**. The three chains are called **input**, **output** and **forward**. When a packet comes in (say, through the Ethernet card) the kernel uses the `input` chain to decide its fate. If it survives that step, then the kernel decides where to send the packet next (this is called **routing**). If it is destined for another machine, it consults the `forward` chain. Finally, just before a packet is to go out, the kernel consults the `output` chain.

A chain is a checklist of **rules**. Each rule says `if the packet header looks like this, then here's what to do with the packet'. If the rule doesn't match the packet, then the next rule in the chain is consulted. Finally, if there are no more rules to consult, then the kernel looks at the chain **policy** to decide what to do. In a security−conscious system, this policy usually tells the kernel to reject or deny the packet.

For ASCII−art fans, this shown the complete path of a packet coming into a machine.

```
        ----------------------------------------------------------------
        |            ACCEPT/                              lo interface |
        v           REDIRECT                  _______                  |
--> C --> S --> ______ --> D --> ~~~~~~~~ -->|forward|----> _______ -->
    h     a    |input |    e    {Routing }   |Chain  |     |output |ACCEPT
    e     n    |Chain |    m    {Decision}   |_______| --->|Chain  |
    c     i    |______|    a     ~~~~~~~~        |     | ->|_______|
    k     t       |        s       |             |     | |     |
    s     y       |        q       |             v     | |     |
    u     |       v        e       v            DENY/  | |     v
    m     |     DENY/      r   Local Process   REJECT  | |   DENY/
    |     v    REJECT      a       |                   | |  REJECT
    |   DENY               d       --------------------- |
    v                      e -----------------------------
   DENY
```

Here is a blow−by−blow description of each stage:

Checksum:

This is a test that the packet hasn't been corrupted in some way. If it has, it is denied.

Sanity:

There is actually one of these sanity checks before each firewall chain, but the input chain's is the most important. Some malformed packets might confuse the rule−checking code, and these are denied here (a message is printed to the syslog if this happens).

input chain:

This is the first firewall chain against which the packet will be tested. If the verdict of the chain is not `DENY` or `REJECT`, the packet continues on.

Demasquerade:

If the packet is a reply to a previously masqueraded packet, it is demasqueraded, and skips straight to the `output` chain. If you don't use IP Masquerading, you can mentally erase this from the diagram.

Routing decision:

The destination field is examined by the routing code, to decide if this packet should go to a local process (see Local process below) or forwarded to a remote machine (see forward chain below).

Local process:

A process running on the machine can receive packets after the Routing Decision step, and can send packets (which go through the Routing Decision step, then traverse the output chain).

lo interface:

If packets from a local process are destined for a local process, they will go through the output chain with interface set to `lo', then return through the input chain with interface also `lo'. The lo interface is usually called the loopback interface.

local:

If the packet was not created by a local process, then the forward chain is checked, otherwise the packet goes to the output chain.

forward chain:

This chain is traversed for any packets which are attempting to pass through this machine to another.

output chain:

This chain is traversed for all packets just before they are sent out.

Using ipchains

First, check that you have the version of ipchains that this document refers to:

```
$ ipchains --version
ipchains 1.3.9, 17-Mar-1999
```

Note that I recommend 1.3.4 (which has no long options, like `−−sport'), or 1.3.8 or above; these are very stable.

ipchains has a fairly detailed manual page (`man ipchains`), and if you need more detail on particulars, you can check out the programming interface (`man 4 ipfw`), or the file `net/ipv4/ip_fw.c` in the 2.1.x kernel source, which is (obviously) authoritative.

There is also an excellent quick reference card by Scott Bronson in the source package, in both A4 and US Letter PostScript(TM).

There are several different things you can do with `ipchains`. First the operations to manage whole chains. You start with three built−in chains `input`, `output` and `forward` which you can't delete.

1. Create a new chain (−N).
2. Delete an empty chain (−X).
3. Change the policy for a built−in chain. (−P).
4. List the rules in a chain (−L).
5. Flush the rules out of a chain (−F).
6. Zero the packet and byte counters on all rules in a chain (−Z).

There are several ways to manipulate rules inside a chain:

1. Append a new rule to a chain (−A).
2. Insert a new rule at some position in a chain (−I).
3. Replace a rule at some position in a chain (−R).
4. Delete a rule at some position in a chain (−D).
5. Delete the first rule that matches in a chain (−D).

There are a few operations for masquerading, which are in `ipchains` for want of a good place to put them:

1. List the currently masqueraded connections (−M −L).
2. Set masquerading timeout values (−M −S). (But see I can't set masquerading timeouts!).

The final (and perhaps the most useful) function allows you to check what would happen to a given packet if it were to traverse a given chain.

What You'll See When Your Computer Starts Up

Before any ipchains commands have been run (be careful: some distributions run ipchains in their initialization scripts), there will be no rules in any of the built−in chains (`input', `forward' and `output'), and each of the chains will have a policy of ACCEPT. This is as wide−open as you can get.

Operations on a Single Rule

This is the bread−and−butter of ipchains; manipulating rules. Most commonly, you will probably use the append (−A) and delete (−D) commands. The others (−I for insert and −R for replace) are simple extensions of these concepts.

Each rule specifies a set of conditions the packet must meet, and what to do if it meets them (a `target'). For example, you might want to deny all ICMP packets coming from the IP address 127.0.0.1. So in this case our conditions are that the protocol must be ICMP and that the source address must be 127.0.0.1. Our target is `DENY'.

127.0.0.1 is the `loopback' interface, which you will have even if you have no real network connection. You can use the `ping' program to generate such packets (it simply sends an ICMP type 8 (echo request) which all cooperative hosts should obligingly respond to with an ICMP type 0 (echo reply) packet). This makes it useful for testing.

```
# ping -c 1 127.0.0.1
PING 127.0.0.1 (127.0.0.1): 56 data bytes
64 bytes from 127.0.0.1: icmp_seq=0 ttl=64 time=0.2 ms

--- 127.0.0.1 ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 0.2/0.2/0.2 ms
# ipchains -A input -s 127.0.0.1 -p icmp -j DENY
# ping -c 1 127.0.0.1
PING 127.0.0.1 (127.0.0.1): 56 data bytes

--- 127.0.0.1 ping statistics ---
1 packets transmitted, 0 packets received, 100% packet loss
#
```

You can see here that the first ping succeeds (the `−c 1' tells ping to only send a single packet).

Then we append (−A) to the `input' chain, a rule specifying that for packets from 127.0.0.1 (`−s 127.0.0.1') with protocol ICMP (`−p ICMP') we should jump to DENY (`−j DENY').

Then we test our rule, using the second ping. There will be a pause before the program gives up waiting for a response that will never come.

We can delete the rule in one of two ways. Firstly, since we know that it is the only rule in the input chain, we can use a numbered delete, as in:

```
# ipchains -D input 1
#
```

To delete rule number 1 in the input chain.

The second way is to mirror the −A command, but replacing the −A with −D. This is useful when you have a complex chain of rules and you don't want to have to count them to figure out that it's rule 37 that you want to get rid of. In this case, we would use:

```
# ipchains -D input -s 127.0.0.1 -p icmp -j DENY
#
```

The syntax of −D must have exactly the same options as the −A (or −I or −R) command. If there are multiple identical rules in the same chain, only the first will be deleted.

Filtering Specifications

We have seen the use of `−p' to specify protocol, and `−s' to specify source address, but there are other options we can use to specify packet characteristics. What follows is an exhaustive compendium.

Specifying Source and Destination IP Addresses

Source (−s) and destination (−d) IP addresses can be specified in four ways. The most common way is to use the full name, such as `localhost' or `www.linuxhq.com'. The second way is to specify the IP address such as `127.0.0.1'.

The third and fourth ways allow specification of a group of IP addresses, such as `199.95.207.0/24' or `199.95.207.0/255.255.255.0'. These both specify any IP address from 199.95.207.0 to 199.95.207.255 inclusive; the digits after the `/' tell which parts of the IP address are significant. `/32' or `/255.255.255.255' is the default (match all of the IP address). To specify any IP address at all `/0' can be used, like so:

```
# ipchains -A input -s 0/0 -j DENY
#
```

This is rarely used, as the effect above is the same as not specifying the `−s' option at all.

Specifying Inversion

Many flags, including the `−s' and `−d' flags can have their arguments preceded by `!' (pronounced `not') to match addresses NOT equal to the ones given. For example. `−s ! localhost' matches any packet not coming from localhost.

Don't forget the spaces around the `!': they really are needed.

Specifying Protocol

The protocol can be specified with the `−p' flag. Protocol can be a number (if you know the numeric protocol values for IP) or a name for the special cases of `TCP', `UDP' or `ICMP'. Case doesn't matter, so `tcp' works as well as `TCP'.

The protocol name can be prefixed by a `!', to invert it, such as `−p ! TCP'.

Specifying UDP and TCP Ports

For the special case where a protocol of TCP or UDP is specified, there can be an extra argument indicating the TCP or UDP port, or an (inclusive) range of ports (but see Handling Fragments below). A range is represented using a `:' character, such as `6000:6010', which covers 11 port numbers, from 6000 to 6010 inclusive. If the lower bound is omitted, it defaults to 0. If the upper bound is omitted, it defaults to 65535. So to specify TCP connections coming from ports under 1024, the syntax would be as `−p TCP −s 0.0.0.0/0 :1023'. Port numbers can be specified by name, eg. `www'.

Note that the port specification can be preceded by a `!', which inverts it. So to specify every TCP packet BUT a WWW packet, you would specify

```
-p TCP -d 0.0.0.0/0 ! www
```

It is important to realize that the specification

```
-p TCP -d ! 192.168.1.1 www
```

is very different from

```
-p TCP -d 192.168.1.1 ! www
```

The first specifies any TCP packet to the WWW port on any machine but 192.168.1.1. The second specifies any TCP connection to any port on 192.168.1.1 but the WWW port.

Finally, this case means not the WWW port and not 192.168.1.1:

```
-p TCP -d ! 192.168.1.1 ! www
```

Specifying ICMP Type and Code

ICMP also allows an optional argument, but as ICMP doesn't have ports, (ICMP has a **type** and a **code**) they have a different meaning.

You can specify them as ICMP names (use `ipchains -h icmp` to list the names) after the `−s' option, or as a numeric ICMP type and code, where the type follows the `−s' option and the code follows the `−d' option.

The ICMP names are fairly long: you only need use enough letters to make the name distinct from any other.

Here is a small table of some of the most common ICMP packets:

```
Number  Name                     Required by

0       echo-reply               ping
3       destination-unreachable  Any TCP/UDP traffic.
5       redirect                 routing if not running routing daemon
8       echo-request             ping
11      time-exceeded            traceroute
```

Note that the ICMP names cannot be preceeded by `!' at the moment.

DO NOT DO NOT DO NOT block all ICMP type 3 messages! (See ICMP Packets below).

Specifying an Interface

The `−i' option specifies the name of an **interface** to match. An interface is the physical device the packet came in on, or is going out on. You can use the `ifconfig` command to list the interfaces which are `up' (ie. working at the moment).

The interface for incoming packets (ie. packets traversing the `input` chain) is considered to be the interface they came in on. Logically, the interface for outgoing packets (packets traversing the `output` chain) is the interface they will go out on. The interface for packets traversing the `forward` chain is also the interface they will go out on; a fairly arbitrary decision it seems to me.

It is perfectly legal to specify an interface that currently does not exist; the rule will not match anything until the interface comes up. This is extremely useful for dial−up PPP links (usually interface `ppp0`) and the like.

As a special case, an interface name ending with a `+' will match all interfaces (whether they currently exist or not) which begin with that string. For example, to specify a rule which matches all PPP interfaces, the `-i ppp+` option would be used.

The interface name can be preceded by a `!' to match a packet which does NOT match the specified interface(s).

Specifying TCP SYN Packets Only

It is sometimes useful to allow TCP connections in one direction, but not the other. For example, you might want to allow connections to an external WWW server, but not connections from that server.

The naive approach would be to block TCP packets coming from the server. Unfortunately, TCP connections require packets going in both directions to work at all.

The solution is to block only the packets used to request a connection. These packets are called **SYN** packets (ok, technically they're packets with the SYN flag set, and the FIN and ACK flags cleared, but we call them SYN packets). By disallowing only these packets, we can stop attempted connections in their tracks.

The `−y' flag is used for this: it is only valid for rules which specify TCP as their protocol. For example, to specify TCP connection attempts from 192.168.1.1:

```
−p TCP −s 192.168.1.1 −y
```

Once again, this flag can be inverted by preceding it with a `!', which means every packet other than the connection initiation.

Handling Fragments

Sometimes a packet is too large to fit down a wire all at once. When this happens, the packet is divided into **fragments**, and sent as multiple packets. The other end reassembles the fragments to reconstruct the whole packet.

The problem with fragments is that some of the specifications listed above (in particular, source port, destinations port, ICMP type, ICMP code, or TCP SYN flag) require the kernel to peek at the start of the packet, which is only contained in the first fragment.

If your machine is the only connection to an external network, then you can tell the Linux kernel to reassemble all fragments which pass through it, by compiling the kernel with `IP: always defragment` set to `Y'. This sidesteps the issue neatly.

Otherwise, it is important to understand how fragments get treated by the filtering rules. Any filtering rule that asks for information we don't have will *not* match. This means that the first fragment is treated like any other packet. Second and further fragments won't be. Thus a rule `−p TCP −s 192.168.1.1 www` (specifying a source port of `www') will never match a fragment (other than the first fragment). Neither will the opposite rule `−p TCP −s 192.168.1.1 ! www`.

However, you can specify a rule specifically for second and further fragments, using the `−f' flag. Obviously, it is illegal to specify a TCP or UDP port, ICMP type, ICMP code or TCP SYN flag in such a fragment rule.

It is also legal to specify that a rule does *not* apply to second and further fragments, by preceding the `−f' with `!'.

Usually it is regarded as safe to let second and further fragments through, since filtering will effect the first fragment, and thus prevent reassembly on the target host, however, bugs have been known to allow crashing of machines simply by sending fragments. Your call.

Note for network−heads: malformed packets (TCP, UDP and ICMP packets too short for the firewalling code to read the ports or ICMP code and type) are treated as fragments as well. Only TCP fragments starting at position 8 are explicitly dropped by the firewall code (a message should appear in the syslog if this occurs).

As an example, the following rule will drop any fragments going to 192.168.1.1:

```
# ipchains −A output −f −d 192.168.1.1 −j DENY
#
```

Filtering Side Effects

OK, so now we know all the ways we can match a packet using a rule. If a packet matches a rule, the following things happen:

1. The byte counter for that rule is increased by the size of the packet (header and all).
2. The packet counter for that rule is incremented.
3. If the rule requests it, the packet is logged.
4. If the rule requests it, the packet's Type Of Service field is changed.
5. If the rule requests it, the packet is marked (not in 2.0 kernel series).
6. The rule target is examined to decide what to do to the packet next.

For variety, I'll address these in order of importance.

Specifying a Target

A **target** tells the kernel what to do with a packet that matches a rule. ipchains uses `−j' (think `jump−to') for the target specification. The target name must be less than 8 letters, and case matters: "RETURN" and "return" are completely different.

The simplest case is when there is no target specified. This type of rule (often called an `accounting' rule) is useful for simply counting a certain type of packet. Whether this rule matches or not, the kernel simply examines the next rule in the chain. For example, to count the number of packets from 192.168.1.1, we could do this:

```
# ipchains −A input −s 192.168.1.1
#
```

(Using `ipchains −L −v' we can see the byte and packet counters associated with each rule).

There are six special targets. The first three, `ACCEPT`, `REJECT` and `DENY` are fairly simple. `ACCEPT` allows the packet through. `DENY` drops the packet as if it had never been received. `REJECT` drops the packet, but (if it's not an ICMP packet) generates an ICMP reply to the source to tell it that the destination was unreachable.

The next one, `MASQ` tells the kernel to masquerade the packet. For this to work, your kernel needs to be compiled with IP Masquerading enabled. For details on this, see the Masquerading−HOWTO and the Appendix Differences between ipchains and ipfwadm. This target is only valid for packets traversing the `forward` chain.

The other major special target is `REDIRECT` which tells the kernel to send a packet to a local port instead of wherever it was heading. This can only be specified for rules specifying TCP or UDP as their protocol. Optionally, a port (name or number) can be specified following `−j REDIRECT' which will cause the packet to be redirected to that particular port, even if it was addressed to another port. This target is only valid for packets traversing the `input` chain.

The final special target is `RETURN` which is identical to falling off the end of the chain immediately. (See Setting Policy below).

Any other target indicates a user−defined chain (as described in Operations on an Entire Chain below). The packet will begin traversing the rules in that chain. If that chain doesn't decide the fate of the packet, then once traversal on that chain has finished, traversal resumes on the next rule in the current chain.

Time for more ASCII art. Consider two (silly) chains: `input` (the built−in chain) and `Test` (a user−defined chain).

```
 `input'                         `Test'
 ----------------------------    ----------------------------
 | Rule1: −p ICMP −j REJECT |    | Rule1: −s 192.168.1.1    |
 |--------------------------|    |--------------------------|
 | Rule2: −p TCP −j Test    |    | Rule2: −d 192.168.1.1    |
 |--------------------------|    ----------------------------
 | Rule3: −p UDP −j DENY    |
 ----------------------------
```

Consider a TCP packet coming from 192.168.1.1, going to 1.2.3.4. It enters the `input` chain, and gets tested against Rule1 − no match. Rule2 matches, and its target is `Test`, so the next rule examined is the start of `Test`. Rule1 in `Test` matches, but doesn't specify a target, so the next rule is examined, Rule2. This doesn't match, so we have reached the end of the chain. We return to the `input` chain, where we had just examined Rule2, so we now examine Rule3, which doesn't match either.

So the packet path is:

```
                        v    __________________________
 `input'                |   /    `Test'                v
------------------------|--/    -----------------------|----
| Rule1                 | /|    | Rule1                |   |
|-----------------------|/-|    |----------------------|---|
| Rule2                 /  |    | Rule2                |   |
|--------------------------|    -----------------------v----
| Rule3                 /--+___________________________/
------------------------|---
                        v
```

See the section How to Organise Your Firewall Rules for ways to use user−defined chains effectively.

Logging Packets

This is a side effect that matching a rule can have; you can have the matching packet logged using the `−l' flag. You will usually not want this for routine packets, but it is a useful feature if you want to look for exceptional events.

The kernel logs this information looking like:

```
Packet log: input DENY eth0 PROTO=17 192.168.2.1:53 192.168.1.1:1025
  L=34 S=0x00 I=18 F=0x0000 T=254
```

This log message is designed to be terse, and contain technical information useful only to networking gurus, but it can be useful to the rest of us. It breaks down like so:

1. `input' is the chain which contained the rule which matched the packet, causing the log message.
2. `DENY' is what the rule said to do to the packet. If this is `−' then the rule didn't effect the packet at all (an accounting rule).
3. `eth0' is the interface name. Because this was the input chain, it means that the packet came in `eth0'.
4. `PROTO=17' means that the packet was protocol 17. A list of protocol numbers is given in `/etc/protocols'. The most common are 1 (ICMP), 6 (TCP) and 17 (UDP).
5. `192.168.2.1' means that the packet's source IP address was 192.168.2.1.
6. `:53' means that the source port was port 53. Looking in `/etc/services' shows that this is the `domain' port (ie. this is probably an DNS reply). For UDP and TCP, this number is the source port. For ICMP, it's the ICMP type. For others, it will be 65535.

7. `192.168.1.1' is the destination IP address.
8. `:1025' means that the destination port was 1025. For UDP and TCP, this number is the destination port. For ICMP, it's the ICMP code. For others, it will be 65535.
9. `L=34' means that packet was a total of 34 bytes long.
10. `S=0x00' means the Type of Service field (divide by 4 to get the Type of Service as used by ipchains).
11. `I=18' is the IP ID.
12. `F=0x0000' is the 16−bit fragment offset plus flags. A value starting with `0x4' or `0x5' means that the Don't Fragment bit is set. `0x2' or `0x3' means the `More Fragments' bit is set; expect more fragments after this. The rest of the number is the offset of this fragment, divided by 8.
13. `T=254' is the Time To Live of the packet. One is subtracted from this value for every hop, and it usually starts at 15 or 255.
14. `(#5)' there may be a final number in brackets on more recent kernels (perhaps after 2.2.9). This is the rule number which caused the packet log.

On standard Linux systems, this kernel output is captured by klogd (the kernel logging daemon) which hands it to syslogd (the system logging daemon). The `/etc/syslog.conf' controls the behaviour of syslogd, by specifying a destination for each `facility' (in our case, the facility is "kernel") and `level' (for ipchains, the level used is "info").

For example, my (Debian) /etc/syslog.conf contains two lines which match `kern.info':

```
kern.*                          -/var/log/kern.log
*.=info;*.=notice;*.=warn;\
        auth,authpriv.none;\
        cron,daemon.none;\
        mail,news.none          -/var/log/messages
```

These mean that the messags are duplicated in `/var/log/kern.log' and `/var/log/messages'. For more details, see `man syslog.conf'.

Manipulating the Type Of Service

There are four seldom−used bits in the IP header, called the **Type of Service** (TOS) bits. They effect the way packets are treated; the four bits are "Minimum Delay", "Maximum Throughput", "Maximum Reliability" and "Minimum Cost". Only one of these bits is allowed to be set. Rob van Nieuwkerk, the author of the TOS−mangling code, puts it as follows:

> Especially the "Minimum Delay" is important for me. I switch it on for "interactive" packets in my upstream (Linux) router. I'm behind a 33k6 modem link. Linux prioritizes packets in 3 queues. This way I get acceptable interactive performance while doing bulk downloads at the same time. (It could even be better if there wasn't such a big queue in the serial driver, but latency is kept down 1.5 seconds now).

Note: obviously, you have no control over incoming packets; you can only control the priority of packets leaving your box. To negotiate priorities with the other end, a protocol like RSVP (which I know nothing about, so don't ask me) must be used.

The most common use is to set telnet & ftp control connections to "Minimum Delay" and FTP data to "Maximum Throughput". This would be done as follows:

```
ipchains −A output −p tcp −d 0.0.0.0/0 telnet −t 0x01 0x10
ipchains −A output −p tcp −d 0.0.0.0/0 ftp −t 0x01 0x10
ipchains −A output −p tcp −s 0.0.0.0/0 ftp−data −t 0x01 0x08
```

The `−t' flag takes two extra parameters, both in hexadecimal. These allow complex twiddling of the TOS bits: the first mask is ANDed with the packet's current TOS, and then the second mask is XORed with it. If this is too confusing, just use the following table:

```
TOS Name                Value           Typical Uses

Minimum Delay           0x01 0x10       ftp, telnet
Maximum Throughput      0x01 0x08       ftp−data
Maximum Reliability     0x01 0x04       snmp
Minimum Cost            0x01 0x02       nntp
```

Andi Kleen goes on to point out the following (mildly edited for posterity):

> Maybe it would be useful to add an reference to the txqueuelen parameter of ifconfig to the discussion of TOS bits. The default device queue length is tuned for ethernet cards, on modems it is too long and makes the 3 band scheduler (which queues based on TOS) work suboptimally. It is a good idea to set it to a value between 4−10 on modem or single b channel ISDN links: on bundled devices a longer queue is needed. This is a 2.0 and 2.1 problem, but in 2.1 it is a ifconfig flag (with recent nettools), while in 2.0 it requires source patches in the device drivers to change.

So, to see maximal benifits of TOS manipulation for modem PPP links, do `ifconfig $1 txqueuelen' in your /etc/ppp/ip−up script. The number to use depends on the modem speed and the amount of buffering in the modem; here's Andi setting me straight again:

> The best value for a given configuration needs experiment. If the queues are too short on a router then packets will get dropped. Also of course one gets benefits even without TOS rewriting, just that TOS rewriting helps to give the benefits to non cooperating programs (but all standard linux programs are cooperating).

Marking a Packet

This allows complex and powerful interactions with Alexey Kuznetsov's new Quality of Service implementation, as well as the mark−based forwarding in later 2.1 series kernels. More news as it comes to hand. This option is ignored altogether in the 2.0 kernel series.

Operations on an Entire Chain

A very useful feature of ipchains is the ability to group related rules into chains. You can call the chains whatever you want, as long as the names don't clash with the built−in chains (`input`, `output` and `forward`) or the targets (`MASQ`, `REDIRECT`, `ACCEPT`, `DENY`, `REJECT` or `RETURN`). I suggest avoiding upper−case labels entirely, since I may use these for future extensions. The chain name can be up to 8 characters long.

Creating a New Chain

Let's create a new chain. Because I am such an imaginative fellow, I'll call it `test`.

```
# ipchains -N test
#
```

It's that simple. Now you can put rules in it as detailed above.

Deleting a Chain

Deleting a chain is simple as well.

```
# ipchains -X test
#
```

Why `−X'? Well, all the good letters were taken.

There are a couple of restrictions to deleting chains: they must be empty (see Flushing a Chain below) and they must not be the target of any rule. You can't delete any of the three built−in chains.

Flushing a Chain

There is a simple way of emptying all rules out of a chain, using the `−F' command.

```
# ipchains -F forward
#
```

If you don't specify a chain, then *all* chains will be flushed.

Listing a Chain

You can list all the rules in a chain by using the `−L' command.

```
# ipchains -L input
Chain input (refcnt = 1): (policy ACCEPT)
target     prot opt    source                destination           ports
ACCEPT     icmp -----  anywhere              anywhere              any
# ipchains -L test
Chain test (refcnt = 0):
target     prot opt    source                destination           ports
DENY       icmp -----  localnet/24           anywhere              any
#
```

The `refcnt' listed for `test` is the number of rules which have `test` as their target. This must be zero (and the chain be empty) before this chain can be deleted.

If the chain name is omitted, all chains are listed, even empty ones.

There are three options which can accompany `−L'. The `−n' (numeric) option is very useful as it prevents `ipchains` from trying to lookup the IP addresses, which (if you are using DNS like most people) will cause large delays if your DNS is not set up properly, or you have filtered out DNS requests. It also causes ports to be printed out as numbers rather than names.

The `−v' options shows you all the details of the rules, such as the the packet and byte counters, the TOS masks, the interface, and the packet mark. Otherwise these values are omitted. For example:

```
# ipchains -v -L input
Chain input (refcnt = 1): (policy ACCEPT)
 pkts bytes target     prot opt   tosa tosx  ifname    mark        source                destination           ports
```

```
   10   840 ACCEPT     icmp ----- 0xFF 0x00  lo                    anywhere              anywhere              any
```

Note that the packet and byte counters are printed out using the suffixes `K', `M' or `G' for 1000, 1,000,000 and 1,000,000,000 respectively. Using the `−x' (expand numbers) flag as well prints the full numbers, no matter how large they are.

Resetting (Zeroing) Counters

It is useful to be able to reset the counters. This can be done with the `−Z' (zero counters) option. For example:

```
# ipchains -v -L input
Chain input (refcnt = 1): (policy ACCEPT)
 pkts bytes target     prot opt   tosa tosx  ifname    mark        source                destination           ports
   10   840 ACCEPT     icmp ----- 0xFF 0x00  lo                    anywhere              anywhere              any
# ipchains -Z input
# ipchains -v -L input
Chain input (refcnt = 1): (policy ACCEPT)
 pkts bytes target     prot opt   tosa tosx  ifname    mark        source                destination           ports
    0     0 ACCEPT     icmp ----- 0xFF 0x00  lo                    anywhere              anywhere              any
#
```

The problem with this approach is that sometimes you need to know the counter values immediately before they are reset. In the above example, some packets could pass through between the `−L' and `−Z' commands. For this reason, you can use the `−L' and `−Z' *together*, to reset the counters while reading them. Unfortunately, if you do this, you can't operate on a single chain: you have to list and zero all the chains at once.

```
# ipchains -L -v -Z
Chain input (policy ACCEPT):
 pkts bytes target     prot opt   tosa tosx  ifname    mark        source                destination           ports
   10   840 ACCEPT     icmp ----- 0xFF 0x00  lo                    anywhere              anywhere              any

Chain forward (refcnt = 1): (policy ACCEPT)
Chain output (refcnt = 1): (policy ACCEPT)
Chain test (refcnt = 0):
    0     0 DENY       icmp ----- 0xFF 0x00  ppp0                  localnet/24           anywhere              any
# ipchains -L -v
Chain input (policy ACCEPT):
 pkts bytes target     prot opt   tosa tosx  ifname    mark        source                destination           ports
   10   840 ACCEPT     icmp ----- 0xFF 0x00  lo                    anywhere              anywhere              any

Chain forward (refcnt = 1): (policy ACCEPT)
Chain output (refcnt = 1): (policy ACCEPT)
Chain test (refcnt = 0):
    0     0 DENY       icmp ----- 0xFF 0x00  ppp0                  localnet/24           anywhere              any
#
```

Setting Policy

We glossed over what happens when a packet hits the end of a built−in chain when we discussed how a packet walks through chains in Specifying a Target above. In this case, the **policy** of the chain determines the fate of the packet. Only built−in chains (`input`, `output` and `forward`) have policies, because if a packet falls off the end of a user−defined chain, traversal resumes at the previous chain.

The policy can be any of the first four special targets: `ACCEPT`, `DENY`, `REJECT` or `MASQ`. `MASQ` is only valid for the `forward' chain.

It is also important to note that a `RETURN` target in a rule in one of the built−in chains is useful to explicitly target the chain policy when a packet matches a rule.

Operations on Masquerading

There are several parameters you can tweak for IP Masquerading. They are bundled with `ipchains` because it's not worth writing a separate tool for them (although this will change).

The IP Masquerading command is `−M', and it can be combined with `−L' to list currently masqueraded connections, or `−S' to set the masquerading parameters.

The `−L' command can be accompanied by `−n' (show numbers instead of hostnames and port names) or `−v' (show deltas in sequence numbers for masqueraded connection, just in case you care).

The `−S' command should be followed by three timeout values, each in seconds: for TCP sessions, for TCP sessions after a FIN packet, and for UDP packets. If you don't want to change one of these values, simply give a value of `0'.

The default values are listed in `/usr/src/linux/include/net/ip_masq.h', currently 15 minutes, 2 minutes and 5 minutes respectively.

The most common value to change is the first one, for FTP (see FTP Nightmares below).

Note the problems with setting timeouts listed in I can't set masquerading timeouts!.

Checking a Packet

Sometimes you want to see what happens when a certain packet enters your machine, such as for debugging your firewall chains. `ipchains` has the `−C' command to allow this, using the exact same routines that the kernel uses to diagnose real packets.

You specify which chain to test the packet on by following the `−C' argument with its name. Whereas the kernel always starts traversing on the `input`, `output` or `forward` chains, you are allowed to begin traversing on any chain for testing purposes.

The details of the `packet' are specified using the same syntax used to specify firewall rules. In particular, a protocol (`−p'), source address (`−s'), destination address (`−d') and interface (`−i') are compulsory. If the protocol is TCP or UDP, then a single source and a single destination port must be specified, and a ICMP type and code must be specified for the ICMP protocol (unless the `−f' flag is specified to indicate a fragment rule, in which case these options are illegal).

If the protocol is TCP (and the `−f' flag is not specified), the `−y' flag may be specified, to indicate that the test packet should have the SYN bit set.

Here is an example of testing a TCP SYN packet from 192.168.1.1 port 60000 to 192.168.1.2 port www, coming in the eth0 interface, entering the `input' chain. (This is a classic incoming WWW connection initiation):

```
# ipchains −C input −p tcp −y −i eth0 −s 192.168.1.1 60000 −d 192.168.1.2 www
packet accepted
#
```

Multiple Rules at Once and Watching What Happens

Sometimes a single command line can result in multiple rules being effected. This is done in two ways. Firstly, if you specify a hostname which resolves (using DNS) to multiple IP addresses, `ipchains` will act as if you had typed multiple commands with each combination of addresses.

So if the hostname `www.foo.com' resolves to three IP addresses, and the hostname `www.bar.com' resolves to two IP addresses, then the command `ipchains −A input −j reject −s www.bar.com −d www.foo.com' would append six rules to the `input` chain.

The other way to have `ipchains` perform multiple actions is to use the bidirectional flag (`−b'). This flag makes `ipchains` behave as if you had typed the command twice, the second time with the `−s' and `−d' arguments reversed. So, to avoid forwarding either to or from 192.168.1.1, you could do the following:

```
# ipchains −b −A forward −j reject −s 192.168.1.1
#
```

Personally, I don't like the `−b' option much; if you want convenience, see Using ipchains−save below.

The −b option can be used with the insert (`−I'), delete (`−D') (but not the variation which takes a rule number), append (`−A') and check (`−C') commands.

Another useful flag is `−v' (verbose) which prints out exactly what `ipchains` is doing with your commands. This is useful if you are dealing with commands that may effect multiple rules. For example, here we check the behaviour of fragments between 192.168.1.1 and 192.168.1.2.

```
# ipchains -v -b -C input -p tcp -f -s 192.168.1.1 -d 192.168.1.2 -i lo
  tcp opt   ---f- tos 0xFF 0x00  via lo    192.168.1.1  -> 192.168.1.2    * ->   *
packet accepted
  tcp opt   ---f- tos 0xFF 0x00  via lo    192.168.1.2  -> 192.168.1.1    * ->   *
packet accepted
#
```

4.2 Useful Examples

I have a dialup PPP connection (`-i ppp0`). I grab news (`-p TCP -s news.virtual.net.au nntp`) and mail (`-p TCP -s mail.virtual.net.au pop-3`) every time I dial up. I use Debian's FTP method to update my machine regularly (`-p TCP -y -s ftp.debian.org.au ftp-data`). I surf the web through my ISP's proxy while this is going on (`-p TCP -d proxy.virtual.net.au 8080`), but hate the ads from doubleclick.net on the Dilbert Archive (`-p TCP -y -d 199.95.207.0/24` and `-p TCP -y -d 199.95.208.0/24`).

I don't mind people trying to ftp to my machine while I'm online (`-p TCP -d $LOCALIP ftp`), but don't want anyone outside pretending to have an IP address of my internal network (`-s 192.168.1.0/24`). This is commonly called IP spoofing, and there is a better way to protect yourself from it in the 2.1.x kernels and above: see How do I set up IP spoof protection?.

This setup is fairly simple, because there are currently no other boxes on my internal network.

I don't want any local process (ie. Netscape, lynx etc.) to connect to doubleclick.net:

```
# ipchains -A output -d 199.95.207.0/24 -j REJECT
# ipchains -A output -d 199.95.208.0/24 -j REJECT
#
```

Now I want to set priorities on various outgoing packets (there isn't much point in doing it on incoming packets). Since I have a fair number of these rules, it makes sense to put them all in a single chain, called `ppp-out`.

```
# ipchains -N ppp-out
# ipchains -A output -i ppp0 -j ppp-out
#
```

Minimum delay for web traffic & telnet.

```
# ipchains -A ppp-out -p TCP -d proxy.virtual.net.au 8080 -t 0x01 0x10
# ipchains -A ppp-out -p TCP -d 0.0.0.0/0 telnet -t 0x01 0x10
#
```

Low cost for ftp data, nntp, pop−3:

```
# ipchains -A ppp-out -p TCP -d 0.0.0.0/0 ftp-data -t 0x01 0x02
# ipchains -A ppp-out -p TCP -d 0.0.0.0/0 nntp -t 0x01 0x02
# ipchains -A ppp-out -p TCP -d 0.0.0.0/0 pop-3 -t 0x01 0x02
#
```

There are a few restrictions on packets coming in the ppp0 interface: let's create a chain called `ppp−in':

```
# ipchains -N ppp-in
# ipchains -A input -i ppp0 -j ppp-in
#
```

Now, no packets coming in `ppp0` should be claiming a source address of 192.168.1.*, so we log and deny them:

```
# ipchains -A ppp-in -s 192.168.1.0/24 -l -j DENY
#
```

I allow UDP packets in for DNS (I run a caching nameserver which forwards all requests to 203.29.16.1, so I expect DNS replies from them only), incoming ftp, and return ftp−data only (which should only be going to a port above 1023, and not the X11 ports around 6000).

```
# ipchains -A ppp-in -p UDP -s 203.29.16.1 -d $LOCALIP dns -j ACCEPT
# ipchains -A ppp-in -p TCP -s 0.0.0.0/0 ftp-data -d $LOCALIP 1024:5999 -j ACCEPT
# ipchains -A ppp-in -p TCP -s 0.0.0.0/0 ftp-data -d $LOCALIP 6010: -j ACCEPT
# ipchains -A ppp-in -p TCP -d $LOCALIP ftp -j ACCEPT
#
```

I allow TCP reply packets back in

```
# ipchains -A ppp-in -p TCP ! -y -j ACCEPT
#
```

Finally, local−to−local packets are OK:

```
# ipchains -A input -i lo -j ACCEPT
#
```

Now, my default policy on the `input` chain is `DENY`, so everything else gets dropped:

```
# ipchains -P input DENY
#
```

NOTE: I wouldn't set up my chains in this order, as packets might get through while I'm setting up. Safest is usually to set the policy to DENY first, then insert the rules. Of course, if your rules require DNS lookups to resolve hostnames, you could be in trouble.

Using ipchains−save

Setting up firewall chains just the way you want them, and then trying to remember the commands you used so you can do them next time is a pain.

So, `ipchains−save` is a script which reads your current chains setup and saves it to a file. For the moment I'll keep you in suspense with regards to what `ipchains−restore` does.

`ipchains−save` can save a single chain, or all chains (if no chain name is specified). The only option currently permitted is `−v' which prints the rules (to stderr) as they are saved. The policy of the chain is also saved for `input`, `output` and `forward` chains.

```
# ipchains-save > my_firewall
Saving `input'.
Saving `output'.
Saving `forward'.
Saving `ppp-in'.
Saving `ppp-out'.
#
```

Using ipchains−restore

`ipchains−restore` restores chains as saved with `ipchains−save`. It can take two options: `−v' which describes each rule as it is added, and `−f' which forces flushing of user−defined chains if they exist, as described below.

If a user−defined chain is found in the input, `ipchains−restore` checks if that chain already exists. If it does, then you will be prompted whether the chains should be flushed (cleared of all rules) or whether restoring this chain should be skipped. If you specified `−f' on the command line, you will not be prompted; the chain will be flushed.

For example:

```
# ipchains-restore < my_firewall
Restoring `input'.
Restoring `output'.
Restoring `forward'.
Restoring `ppp-in'.
Chain `ppp-in' already exists. Skip or flush? [S/f]? s
Skipping `ppp-in'.
Restoring `ppp-out'.
Chain `ppp-out' already exists. Skip or flush? [S/f]? f
Flushing `ppp-out'.
#
```

5.Miscellaneous.

This section contains all the information and FAQs that I couldn't fit inside the structure above.

5.1 How to Organize Your Firewall Rules

This question requires some thought. You can try to organize them to optimize speed (minimize the number of rule−checks for the most common packets) or to increase manageability.

If you have an intermittent link, say a PPP link, you might want to set the first rule in the input chain to be set to `−i ppp0 −j DENY' at boot time, then have something like this in your `ip−up` script:

```
# Re−create the `ppp−in' chain.
ipchains−restore −f < ppp−in.firewall

# Replace DENY rule with jump to ppp−handling chain.
ipchains −R input 1 −i ppp0 −j ppp−in
```

Your `ip−down` script would look like:

```
ipchains −R input 1 −i ppp0 −j DENY
```

5.2 What Not To Filter Out

There are some things you should be aware of before you start filtering out everything you don't want.

ICMP packets

ICMP packets are used (among other things) to indicate failure for other protocols (such as TCP and UDP). `destination−unreachable' packets in particular. Blocking these packets means that you will never get `Host unreachable' or `No route to host' errors; any connections will just wait for a reply that never comes. This is irritating, but rarely fatal.

A worse problem is the role of ICMP packets in MTU discovery. All good TCP implementations (Linux included) use MTU discovery to try to figure out what the largest packet that can get to a destination without being fragmented (fragmentation slows performance, especially when occasional fragments are lost). MTU discovery works by sending packets with the "Don't Fragment" bit set, and then sending smaller packets if it gets an ICMP packet indicating "Fragmentation needed but DF set" (`fragmentation−needed'). This is a type of `destination−unreachable' packet, and if it is never received, the local host will not reduce MTU, and performance will be abysmal or non−existent.

Note that it is common to block all ICMP redirect messages (type 5); these can be used to manipulate routing (although good IP stacks have safeguards), and so are often seen as slightly risky.

TCP Connections to DNS (nameservers)

If you're trying to block outgoing TCP connections, remember that DNS doesn't always use UDP; if the reply from the server exceeds 512 bytes, the client uses a TCP connection (still going to port number 53) to get the data.

This can be a trap because DNS will `mostly work' if you disallow such TCP transfers; you may experience strange long delays and other occasional DNS problems if you do.

If your DNS queries are always directed at the same external source (either directly by using the `nameserver` line in `/etc/resolv.conf` or by using a caching nameserver in forward mode), then you need only allow TCP connections to port `domain` on that nameserver from the local `domain` port (if using a caching nameserver) or from a high port (> 1023) if using `/etc/resolv.conf`.

FTP Nightmares

The classic packet filtering problem is FTP. FTP has two **modes**; the traditional one is called **active mode** and the more recent one is called **passive mode**. Web browsers usually default to passive mode, but command−line FTP programs usually default to active mode.

In active mode, when the remote end wants to send a file (or even the results of an `ls` or `dir` command) it tries to open a TCP connection to the local machine. This means you can't filter out these TCP connections without breaking active FTP.

If you have the option of using passive mode, then fine; passive mode makes data connections from client to server, even for incoming data. Otherwise, it is recommended that you only allow TCP connections to ports above 1024 and not between 6000 and 6010 (6000 is used for X−Windows).

5.3 Filtering out Ping of Death

Linux boxes are now immune to the famous **Ping of Death**, which involves sending an illegally−large ICMP packet which overflows buffers in the TCP stack on the receiver and causes havoc.

If you are protecting boxes which might be vulnerable, you could simply block ICMP fragments. Normal ICMP packets aren't large enough to require fragmentation, so you won't break anything except big pings. I have heard (unconfirmed) reports that some systems required only the last fragment of an oversize ICMP packet to corrupt them, so blocking only the first fragment is not recommended.

While the exploit programs I have seen all use ICMP, there is no reasons that TCP or UDP fragments (or an unknown protocol) could not be used for this attack, so blocking ICMP fragments is only a temporary solution.

5.4 Filtering out Teardrop and Bonk

Teardrop and Bonk are two attacks (mainly against Microsoft Windows NT machines) which rely on overlapping fragments. Having your Linux router do defragmentation, or disallowing all fragments to your vulnerable machines are the other options.

5.5 Filtering out Fragment Bombs

Some less−reliable TCP stacks are said to have problems dealing with large numbers of fragments of packets when they don't receive all the fragments. Linux does not have this problem. You can filter out fragments (which might break legitimate uses) or compile your kernel with `IP: always defragment' set to `Y' (only if your Linux box is the only possible route for these packets).

5.6 Changing Firewall Rules

There are some timing issues involved in altering firewall rules. If you are not careful, you can let packets through while you are half−way through your changes. A simplistic approach is to do the following:

```
# ipchains -I input 1 -j DENY
# ipchains -I output 1 -j DENY
# ipchains -I forward 1 -j DENY

... make changes ...

# ipchains -D input 1
# ipchains -D output 1
# ipchains -D forward 1
#
```

This drops all packets for the duration of the changes.

If your changes are restricted to a single chain, you might want to create a new chain with the new rules, and then replace (`−R') the rule that pointed to the old chain with one that points to the new chain: then you can delete the old chain. This replacement will occur atomically.

5.7 How Do I Set Up IP Spoof Protection?

IP spoofing is a technique where a host sends out packets which claim to be from another host. Since packet filtering makes decisions based on this source address, IP spoofing is uses to fool packet filters. It is also used to hide the identity of attackers using SYN attacks, Teardrop, Ping of Death and the like (don't worry if you don't know what they are).

The best way to protect from IP spoofing is called Source Address Verification, and it is done by the routing code, and not firewalling at all. Look for a file called `/proc/sys/net/ipv4/conf/all/rp_filter`. If this exists, then turning on Source Address Verification at every boot is the right solution for you. To do that, insert the following lines somewhere in your init scripts, before any network interfaces are initialized:

```
# This is the best method: turn on Source Address Verification and get
# spoof protection on all current and future interfaces.
if [ -e /proc/sys/net/ipv4/conf/all/rp_filter ]; then
  echo -n "Setting up IP spoofing protection..."
  for f in /proc/sys/net/ipv4/conf/*/rp_filter; do
      echo 1 > $f
  done
  echo "done."
else
  echo PROBLEMS SETTING UP IP SPOOFING PROTECTION.  BE WORRIED.
  echo "CONTROL-D will exit from this shell and continue system startup."
  echo
  # Start a single user shell on the console
  /sbin/sulogin $CONSOLE
fi
```

If you cannot do this, you can manually insert rules to protect every interface. This requires knowledge of each interface. The 2.1 kernels automatically reject packets claiming to come from the 127.* addresses (reserved for the local loopback interface, `lo`).

For example, say we have three interfaces, `eth0`, `eth1` and `ppp0`. We can use `ifconfig` to tell us the address and netmask of the interfaces. Say `eth0` was attached to a network 192.168.1.0 with netmask 255.255.255.0, `eth1` was attached to a network 10.0.0.0 with netmask 255.0.0.0, and `ppp0` connected to the Internet (where any address except the reserved private IP addresses are allowed), we would insert the following rules:

```
# ipchains -A input -i eth0 -s ! 192.168.1.0/255.255.255.0 -j DENY
# ipchains -A input -i ! eth0 -s 192.168.1.0/255.255.255.0 -j DENY
# ipchains -A input -i eth1 -s ! 10.0.0.0/255.0.0.0 -j DENY
# ipchains -A input -i ! eth1 -s 10.0.0.0/255.0.0.0 -j DENY
#
```

This approach is not as good as the Source Address Verification approach, because if your network changes, you have to change your firewalling rules to keep up.

If you are running a 2.0 series kernel, you might want to protect the loopback interface as well, using a rule like this:

```
# ipchains -A input -i ! lo -s 127.0.0.0/255.0.0.0 -j DENY
#
```

5.8 Advanced Projects

There is a userspace library I have written which is included with the source distribution called `libfw'. It uses the ability of IP Chains 1.3 and above to copy a packet to userspace (using the IP_FIREWALL_NETLINK config option).

The mark value can be used to specify the Quality of Service parameters for packets, or to specify how packets should be port−forwarded. I've never used either, but if you want to write about it, please contact me.

Things such as **stateful inspection** (I prefer the term dynamic firewalling) can be implemented in userspace using this library. Other nifty ideas include controlling packets on a per−user basis by doing a lookup in a userspace daemon. This should be pretty easy.

SPF: Stateful Packet Filtering

ftp://ftp.interlinx.bc.ca/pub/spf is the site of Brian Murrell's SPF project, which does connection tracking in userspace. It adds significant security for low−bandwidth sites.

There's little documentation at present, but here's a post to the mailing list in which Brian answered some questions:

```
> I believe it does exactly what I want: Installing a temporary
```

```
> "backward"-rule to let packets in as a response to an
> outgoing request.

Yup, that is exactly what it does.  The more protocols it
understands, the more "backward" rules it gets right.  Right
now it has support for (from memory, please excuse any errors
or omissions) FTP (both active and passive, in and out), some
RealAudio, traceroute, ICMP and basic ICQ (inbound from the ICQ
servers, and direct TCP connections, but alas the secondary
direct TCP connections for things like file transfer, etc. are
not there yet)

> Is it a replacement for ipchains or a supplement?

It is a supplement.  Think of ipchains as the engine to allow
and prevent packets from travelling across a Linux box.  SPF is
the driver, constantly monitoring traffic and telling ipchains
how to change it's policies to reflect the changes in traffic
patterns.
```

Michael Hasenstein's ftp−data hack

Michael Hasenstein of SuSE has written a kernel patch which adds ftp connection tracking to ipchains. It can currently be found at http://www.suse.de/~mha/patch.ftp−data−2.gz

5.9 Future Enhancements

Firewalling and NAT have being redesigned for 2.4. Plans and discussions are available on the netfilter list (see http://lists.samba.org). These enhancements should clear up many outstanding usability issues (really, firewalling and masquerading shouldn't be *this hard*), and allow growth for far more flexible firewalling.

6.Common Problems

6.1 ipchains −L Freezes!

You're probably blocking DNS lookups; it will eventually time out. Try using the `−n' (numeric) flag to ipchains, which suppresses the lookup of names.

6.2 Inverse doesn't work!

You must put the `!' option by itself, with spaces either side. A classic mistake (warned about in 1.3.10) is:

```
# ipchains -A input -i !eth0 -j DENY
#
```

There will never be an interface called `!eth0', but ipchains doesn't know that.

6.3 Masquerading/Forwarding Doesn't Work!

Make sure that packet forwarding is enabled (in recent kernels it is disabled by default, meaning that packets never even try to traverse the `forward' chain). You can override this (as root) by typing

```
# echo 1 > /proc/sys/net/ipv4/ip_forward
#
```

If this works for you, you can put this somewhere in your bootup scripts so it is enabled every time; you'll want to set up your firewalling before this command runs though, otherwise there's an opportunity for packets to slip through.

6.4 −j REDIR doesn't work!

You must allow forwarding packets (see above) for redirect to work; otherwise the routing code drops the packet. So if you are just using redirect, and don't have any forwarding at all, you should be aware of that.

Note that REDIR (being in the input chain) doesn't effect connections from a local process.

6.5 Wildcard Interfaces Don't Work!

There was a bug in versions 2.1.102 and 2.1.103 of the kernel (and some old patches I produced) which made ipchains commands which specified a wildcard interface (such as `-i ppp+`) fail.

This is fixed in recent kernels, and in the 2.0.34 patch on the web site. You can also fix it by hand in the kernel source by changing line 63 or so in include/linux/ip_fw.h:

```
#define IP_FW_F_MASK    0x002F  /* All possible flag bits mask   */
```

This should read ``0x003F''. Fix this and recompile the kernel.

6.6 TOS Doesn't Work!

This was my mistake: setting the Type of Service field did not actually set the Type of Service in kernel versions 2.1.102 through 2.1.111. This problem was fixed in 2.1.112.

6.7 ipautofw and ipportfw Don't Work!

For 2.0.x, this is true; I haven't time to create and maintain a jumbo patch for ipchains and ipautofw/ipportfw.

For 2.1.x, download Juan Ciarlante's ipmasqadm from

```
<url url="http://juanjox.linuxhq.com/"
        name="http://juanjox.linuxhq.com/">
```

and use it exactly as you would have used `ipautofw` or `ipportfw`, except instead of `ipportfw` you type `ipmasqadm portfw`, and instead of `ipautofw` you type `ipmasqadm autofw`.

6.8 xosview is Broken!

Upgrade to version 1.6.0 or above, which doesn't require any firewall rules at all for 2.1.x kernels. This seems to have broken again in the 1.6.1 release; please bug the author (it's not my fault!).

6.9 Segmentation Fault With `−j REDIRECT'!

This was a bug in ipchains version 1.3.3. Please upgrade.

6.10 I Can't Set Masquerading Timeouts!

True (for 2.1.x kernels) up to 2.1.123. In 2.1.124, trying to set the masquerading timeouts causes a kernel lockup (change `return` to `ret =` on line 1328 of net/ipv4/ip_fw.c). In 2.1.125, it works fine.

6.11 I Want to Firewall IPX!

So do a number of others, it seems. My code only covers IP, unfortunately. On the good side, all the hooks are there to firewall IPX! You just need to write the code; I will happily help where possible.

7.A Serious Example.

This example was extracted from Michael Neuling and my March 1999 LinuxWorld Tutorial; this is not the only way to solve the given problem, but it is probably the simplest. I hope you will find it informative.

7.1 The Arrangement

- Masqueraded internal network (various operating systems), which we call "GOOD".
- Exposed servers in a separate network (called "DMZ" for Demilitarized Zone).
- PPP Connection to the Internet (called "BAD").

```
External Network (BAD)
        |
        |
    ppp0|
 ---------------
 | 192.84.219.1|             Server Network (DMZ)
 |             |eth0
 |             |----------------------------------------------
 |             |192.84.219.250 |             |              |
 |             |               |             |              |
 |192.168.1.250|               |             |              |
 ---------------          --------       -------        -------
        | eth1            | SMTP |       | DNS |        | WWW |
        |                 --------       -------        -------
        |              192.84.219.128  192.84.219.129  192.84.218.130
        |
Internal Network (GOOD)
```

7.2 Goals

Packet Filter box:

PING any network

This is really useful to tell if a machine is down.

TRACEROUTE any network

Once again, useful for diagnosis.

Access DNS

To make ping and DNS more useful.

Within the DMZ:

Mail server

- SMTP to external
- Accept SMTP from internal and external
- Accept POP−3 from internal

Name Server

- Send DNS to external
- Accept DNS from internal, external and packet filter box

Web server

- Accept HTTP from internal and external
- Rsync access from internal

Internal:

Allow WWW, ftp, traceroute, ssh to external

These are fairly standard things to allow: some places start by allowing the internal machines to do just about everything, but here we're being restrictive.

Allow SMTP to Mail server

Obviously, we want them to be able to send mail out.

Allow POP−3 to Mail server

This is how they read their mail.

Allow DNS to Name server

They need to be able to look up external names for WWW, ftp, traceroute and ssh.

Allow rsync to Web server

This is how they synchronize the external web server with the internal one.

Allow WWW to Web server

Obviously, they should be able to connect to our external web server.

Allow ping to packet filter box

This is a courteous thing to allow: it means that they can test if the firewall box is down (so we don't get blamed if an external site is broken).

7.3 Before Packet Filtering

- Anti−spoofing

Since we don't have any asymmetric routing, we can simply turn on anti−spoofing for all interfaces.

```
# for f in /proc/sys/net/ipv4/conf/*/rp_filter; do echo 1 > $f; done
#
```

- Set filtering rules to DENY all:

We still allow local loopback traffic, but deny anything else.

```
# ipchains -A input -i ! lo -j DENY
# ipchains -A output -i ! lo -j DENY
# ipchains -A forward -j DENY
#
```

- Set Up Interfaces

This is usually done in the boot scripts. Make sure the above steps are done before the interfaces are configured, to prevent packet leakage before the rules are set up.

- Insert per−protocol masquerading modules.

  We need to insert the masquerading module for FTP, so that active and passive FTP `just work' from the internal network.

```
# insmod ip_masq_ftp
#
```

7.4 Packet Filtering for Through Packets

With masquerading, it's best to filter in the forward chain.

Split forward chain into various user chains depending on source/dest interfaces; this breaks the problem down into managable chunks.

```
ipchains −N good−dmz
ipchains −N bad−dmz
ipchains −N good−bad
ipchains −N dmz−good
ipchains −N dmz−bad
ipchains −N bad−good
```

ACCEPTing standard error ICMPs is a common thing to do, so we create a chain for it.

```
ipchains −N icmp−acc
```

Set Up Jumps From forward Chain

Unfortunately, we only know (in the forward chain) the outgoing interface. Thus, to figure out what interface the packet came in on, we use the source address (the anti−spoofing prevents address faking).

Note that we log anything which doesn't match any of these (obviously, this should never happen).

```
ipchains -A forward -s 192.168.1.0/24 -i eth0 -j good-dmz
ipchains -A forward -s 192.168.1.0/24 -i ppp0 -j good-bad
ipchains -A forward -s 192.84.219.0/24 -i ppp0 -j dmz-bad
ipchains -A forward -s 192.84.219.0/24 -i eth1 -j dmz-good
ipchains -A forward -i eth0 -j bad-dmz
ipchains -A forward -i eth1 -j bad-good
ipchains -A forward -j DENY -l
```

Define the icmp−acc Chain

Packets which are one of the error ICMPs get ACCEPTed, otherwise, control will pass back to the calling chain.

```
ipchains -A icmp-acc -p icmp --icmp-type destination-unreachable -j ACCEPT
ipchains -A icmp-acc -p icmp --icmp-type source-quench -j ACCEPT
ipchains -A icmp-acc -p icmp --icmp-type time-exceeded -j ACCEPT
ipchains -A icmp-acc -p icmp --icmp-type parameter-problem -j ACCEPT
```

Good (Internal) to DMZ (Servers)

Internal restrictions:

- Allow WWW, ftp, traceroute, ssh to external
- **Allow SMTP to Mail server**
- **Allow POP−3 to Mail server**
- **Allow DNS to Name server**
- **Allow rsync to Web server**
- **Allow WWW to Web server**
- Allow ping to packet filter box

Could do masquerading from internal network into DMZ, but here we don't. Since noone in the internal network should be trying to do evil things, we log any packets that get denied.

Note that old versions of Debian called `pop3' `pop−3' in /etc/services, which disagrees with RFC1700.

```
ipchains -A good-dmz -p tcp -d 192.84.219.128 smtp -j ACCEPT
ipchains -A good-dmz -p tcp -d 192.84.219.128 pop3 -j ACCEPT
ipchains -A good-dmz -p udp -d 192.84.219.129 domain -j ACCEPT
ipchains -A good-dmz -p tcp -d 192.84.219.129 domain -j ACCEPT
ipchains -A good-dmz -p tcp -d 192.84.218.130 www -j ACCEPT
ipchains -A good-dmz -p tcp -d 192.84.218.130 rsync -j ACCEPT
ipchains -A good-dmz -p icmp -j icmp-acc
ipchains -A good-dmz -j DENY -l
```

Bad (external) to DMZ (servers).

- DMZ restrictions:
  - Mail server
    - **SMTP to external**
    - **Accept SMTP from** internal and **external**
    - Accept POP−3 from internal
  - Name server
    - **Send DNS to external**
    - **Accept DNS from** internal, **external** and packet filter box
  - Web server
    - **Accept HTTP from** internal and **external**
    - Rsync access from internal
- Things we allow from external network to DMZ.
  - Don't log violations, as they may happen.

```
ipchains -A bad-dmz -p tcp -d 192.84.219.128 smtp -j ACCEPT
ipchains -A bad-dmz -p udp -d 192.84.219.129 domain -j ACCEPT
ipchains -A bad-dmz -p tcp -d 192.84.219.129 domain -j ACCEPT
ipchains -A bad-dmz -p tcp -d 192.84.218.130 www -j ACCEPT
ipchains -A bad-dmz -p icmp -j icmp-acc
ipchains -A bad-dmz -j DENY
```

Good (internal) to Bad (external).

- Internal restrictions:
  - **Allow WWW, ftp, traceroute, ssh to external**
  - Allow SMTP to Mail server
  - Allow POP−3 to Mail server
  - Allow DNS to Name server
  - Allow rsync to Web server
  - Allow WWW to Web server
  - Allow ping to packet filter box
- Many people allow everything from the internal to external networks, then add restrictions. We're being fascist.
  - Log violations.
  - Passive FTP handled by masq. module.
  - UDP destination ports 33434 and up are used by traceroute.

```
ipchains -A good-bad -p tcp --dport www -j MASQ
ipchains -A good-bad -p tcp --dport ssh -j MASQ
ipchains -A good-bad -p udp --dport 33434:33500 -j MASQ
ipchains -A good-bad -p tcp --dport ftp -j MASQ
ipchains -A good-bad -p icmp --icmp-type ping -j MASQ
ipchains -A good-bad -j REJECT -l
```

DMZ to Good (internal).

- Internal restrictions:
    - Allow WWW, ftp, traceroute, ssh to external
    - **Allow SMTP to Mail server**
    - **Allow POP−3 to Mail server**
    - **Allow DNS to Name server**
    - **Allow rsync to Web server**
    - **Allow WWW to Web server**
    - Allow ping to packet filter box
- If we were masquerading from the internal network to the DMZ, simply refuse any packets coming the other way. As it is, only allow packets which might be part of an established connection.

```
ipchains -A dmz-good -p tcp ! -y -s 192.84.219.128 smtp -j ACCEPT
ipchains -A dmz-good -p udp -s 192.84.219.129 domain -j ACCEPT
ipchains -A dmz-good -p tcp ! -y -s 192.84.219.129 domain -j ACCEPT
ipchains -A dmz-good -p tcp ! -y -s 192.84.218.130 www -j ACCEPT
ipchains -A dmz-good -p tcp ! -y -s 192.84.218.130 rsync -j ACCEPT
ipchains -A dmz-good -p icmp -j icmp-acc
ipchains -A dmz-good -j DENY -l
```

DMZ to bad (external).

- DMZ restrictions:
    - Mail server
        - **SMTP to external**
        - **Accept SMTP from** internal and **external**
        - Accept POP−3 from internal
    - Name server
        - **Send DNS to external**
        - **Accept DNS from** internal, **external** and packet filter box
    - Web server

- ◊ **Accept HTTP from** internal and **external**
- ◊ Rsync access from internal

•

```
ipchains -A dmz-bad -p tcp -s 192.84.219.128 smtp -j ACCEPT
ipchains -A dmz-bad -p udp -s 192.84.219.129 domain -j ACCEPT
ipchains -A dmz-bad -p tcp -s 192.84.219.129 domain -j ACCEPT
ipchains -A dmz-bad -p tcp ! -y -s 192.84.218.130 www -j ACCEPT
ipchains -A dmz-bad -p icmp -j icmp-acc
ipchains -A dmz-bad -j DENY -l
```

Bad (external) to Good (internal).

- We don't allow anything (non−masqueraded) from the external network to the internal network

```
ipchains -A bad-good -j REJECT
```

Packet Filtering for the Linux Box Itself

- If we want to use packet filtering on packets coming into the box itself, we need to do filtering in the input chain. We create one chain for each destination interface:

```
ipchains -N bad-if
ipchains -N dmz-if
ipchains -N good-if
```

- Create jumps to them:

```
ipchains -A input -d 192.84.219.1 -j bad-if
ipchains -A input -d 192.84.219.250 -j dmz-if
ipchains -A input -d 192.168.1.250 -j good-if
```

Bad (external) interface.

- Packet Filter box:
  - ♦ **PING any network**

- ♦ **TRACEROUTE any network**
- ♦ Access DNS
- External interface also receives replies to masqueraded packets (masquerading uses source ports 61000 to 65095) and ICMP errors for them and PING replies.

```
ipchains -A bad-if -i ! ppp0 -j DENY -l
ipchains -A bad-if -p TCP --dport 61000:65095 -j ACCEPT
ipchains -A bad-if -p UDP --dport 61000:65095 -j ACCEPT
ipchains -A bad-if -p ICMP --icmp-type pong -j ACCEPT
ipchains -A bad-if -j icmp-acc
ipchains -A bad-if -j DENY
```

DMZ interface.

- Packet Filter box restrictions:
  - ♦ **PING any network**
  - ♦ **TRACEROUTE any network**
  - ♦ **Access DNS**
- DMZ interface receives DNS replies, ping replies and ICMP errors.

```
ipchains -A dmz-if -i ! eth0 -j DENY
ipchains -A dmz-if -p TCP ! -y -s 192.84.219.129 53 -j ACCEPT
ipchains -A dmz-if -p UDP -s 192.84.219.129 53 -j ACCEPT
ipchains -A dmz-if -p ICMP --icmp-type pong -j ACCEPT
ipchains -A dmz-if -j icmp-acc
ipchains -A dmz-if -j DENY -l
```

Good (internal) interface.

- Packet Filter box restrictions:
  - ♦ **PING any network**
  - ♦ **TRACEROUTE any network**
  - ♦ **Access DNS**
- Internal restrictions:
  - ♦ Allow WWW, ftp, traceroute, ssh to external
  - ♦ Allow SMTP to Mail server
  - ♦ Allow POP−3 to Mail server
  - ♦ Allow DNS to Name server
  - ♦ Allow rsync to Web server

    - Allow WWW to Web server
    - **Allow ping to packet filter box**
- Internal interface receives pings, ping replies and ICMP errors.

```
ipchains -A good-if -i ! eth1 -j DENY
ipchains -A good-if -p ICMP --icmp-type ping -j ACCEPT
ipchains -A good-if -p ICMP --icmp-type pong -j ACCEPT
ipchains -A good-if -j icmp-acc
ipchains -A good-if -j DENY -l
```

7.5 Finally

- Delete blocking rules:

```
ipchains -D input 1
ipchains -D forward 1
ipchains -D output 1
```

8.Appendix: Differences between ipchains and ipfwadm.

Some of these changes are a result of kernel changes, and some a result of `ipchains` being different from `ipfwadm`.

1. Many arguments have been remapped: capitals now indicates a command, and lower case now indicates an option.
2. Arbitrary chains are supported, so even built−in chains have full names instead of flags (eg. `input' instead of `−I').
3. The `−k' option has vanished: use `! −y'.
4. The `−b' option actually inserts/appends/deletes two rules, rather than a single `bidirectional' rule.
5. The `−b' option can be passed to `−C' to do two checks (one in each direction).
6. The `−x' option to `−l' has been replaced by `−v'.
7. Multiple source and destination ports are not supported anymore. Hopefully being able to negate the port range will somewhat make up for that.
8. Interfaces can only be specified by name (not address). The old semantics got silently changed in the 2.1 kernel series anyway.
9. Fragments are examined, not automatically allowed through.
10. Explicit accounting chains have been done away with.
11. Arbitrary protocols over IP can be tested for.
12. The old behavior of SYN and ACK matching (which was previously ignored for non−TCP packets) has changed; the SYN option is not valid for non−TCP−specific rules.

13. Counters are now 64−bit on 32−bit machines, not 32−bit.
14. Inverse options are now supported.
15. ICMP codes are now supported.
16. Wildcard interfaces are now supported.
17. TOS manipulations are now sanity−checked: the old kernel code would silently stop you from (illegally) manipulating the `Must Be Zero' TOS bit; ipchains now returns an error if you try, as well as for other illegal cases.

8.1 Quick−Reference table.

[Mainly, command arguments are UPPER CASE, and option arguments are lower case]

One thing to note, masquerading is specified by `−j MASQ'; it is completely different from `−j ACCEPT', and not treated as merely a side−effect, unlike `ipfwadm` does.

```
=================================================================
| ipfwadm      | ipchains               | Notes
-----------------------------------------------------------------
| -A [both]    | -N acct                | Create an `acct' chain
|              |& -I 1 input -j acct    | and have output and input
|              |& -I 1 output -j acct   | packets traverse it.
|              |& acct                  |
-----------------------------------------------------------------
| -A in        | input                  | A rule with no target
-----------------------------------------------------------------
| -A out       | output                 | A rule with no target
-----------------------------------------------------------------
| -F           | forward                | Use this as [chain].
-----------------------------------------------------------------
| -I           | input                  | Use this as [chain].
-----------------------------------------------------------------
| -O           | output                 | Use this as [chain].
-----------------------------------------------------------------
| -M -l        | -M -L                  |
-----------------------------------------------------------------
| -M -s        | -M -S                  |
-----------------------------------------------------------------
| -a policy    | -A [chain] -j POLICY   | (but see -r and -m).
-----------------------------------------------------------------
| -d policy    | -D [chain] -j POLICY   | (but see -r and -m).
-----------------------------------------------------------------
| -i policy    | -I 1 [chain] -j POLICY| (but see -r and -m).
-----------------------------------------------------------------
| -l           | -L                     |
-----------------------------------------------------------------
| -z           | -Z                     |
-----------------------------------------------------------------
| -f           | -F                     |
-----------------------------------------------------------------
| -p           | -P                     |
-----------------------------------------------------------------
| -c           | -C                     |
-----------------------------------------------------------------
```

-P	-p	
-S	-s	Only takes one port or range, not multiples.
-D	-d	Only takes one port or range, not multiples.
-V	<none>	Use -i [name].
-W	-i	
-b	-b	Now actually makes 2 rules.
-e	-v	
-k	! -y	Doesn't work unless -p tcp also specified.
-m	-j MASQ	
-n	-n	
-o	-l	
-r [redirpt]	-j REDIRECT [redirpt]	
-t	-t	
-v	-v	
-x	-x	
-y	-y	Doesn't work unless -p tcp also specified.

8.2 Examples of translated ipfwadm commands

Old command: ipfwadm –F –p deny

New command: ipchains –P forward DENY

Old command: ipfwadm –F –a m –S 192.168.0.0/24 –D 0.0.0.0/0

New command: ipchains –A forward –j MASQ –s 192.168.0.0/24 –d 0.0.0.0/0

Old command: ipfwadm –I –a accept –V 10.1.2.1 –S 10.0.0.0/8 –D 0.0.0.0/0

New command: ipchains –A input –j ACCEPT –i eth0 –s 10.0.0.0/8 –d 0.0.0.0/0

(Note that there is no equivalent for specifying interfaces by address: use the interface name. On this machine, 10.1.2.1 corresponds to eth0).

9.Appendix: Using the ipfwadm−wrapper script.

The `ipfwadm−wrapper` shell script should be a plug−in replacement of `ipfwadm` for backwards compatibility with ipfwadm 2.3a.

The only feature it can't really handle is the `−V' option. When this is used, a warning is given. If the `−W' option is also used, the `−V' option is ignored. Otherwise, the script tries to find the interface name associated with that address, using `ifconfig`. If that fails (such as for an interface which is down) then it will exit with an error message.

This warning can be suppressed by either changing the `−V' to a `−W', or directing the standard output of the script to /dev/null.

If you should find any mistakes in this script, or any changes between the real ipfwadm and this script, *please* report a bug to me: send an EMail to rusty@linuxcare.com with "BUG−REPORT" in the subject. Please list your old version of `ipfwadm` (`ipfwadm −h`), your version of `ipchains` (`ipchains --version`), the version of the ipfwadm wrapper script (`ipfwadm−wrapper --version`). Also send the output of `ipchains−save`. Thanks in advance.

Mix `ipchains` with this `ipfwadm−wrapper` script at your own peril.

10.Appendix: Thanks.

Many thanks have to go to Michael Neuling, who wrote the first releasable cut of the IP chains code while working for me. Public apologies for nixing his result−caching idea, which Alan Cox later proposed and I have finally begun implementing, having seen the error of my ways.

Thanks to Alan Cox for his 24−hour EMail tech support, and encouragement.

Thanks to all the authors of the ipfw and ipfwadm code, especially Jos Vos. Standing on the shoulders of giants and all that... This applies to Linus Torvalds and all the kernel and userspace hackers as well.

Thanks to the diligent beta testers and bughunters, especially Jordan Mendelson, Shaw Carruthers, Kevin Moule, Dr. Liviu Daia, Helmut Adams, Franck Sicard, Kevin Littlejohn, Matt Kemner, John D. Hardin, Alexey Kuznetsov, Leos Bitto, Jim Kunzman, Gerard Gerritsen, Serge Sivkov, Andrew Burgess, Steve Schmidtke, Richard Offer, Bernhard Weisshuhn, Larry Auton, Ambrose Li, Pavel Krauz, Steve Chadsey, Francesco Potorti`, Alain Knaff, Casper Boden−Cummins and Henry Hollenberg.

10.1 Translations

People who do translations should put themselves at the *top* of the Thanks page, like so: `Special thanks to XXX, for translating everything exactly from my English.'. Then tell me about your translation so I can include it here.

Arnaud Launay, asl@launay.org: http://www.freenix.fr/unix/linux/HOWTO/IPCHAINS−HOWTO.html

Giovanni Bortolozzo, borto@pluto.linux.it:
http://www.pluto.linux.it/ildp/HOWTO/IPCHAINS−HOWTO.html

Herman Rodríguez, herman@maristas.dhis.org:
http://netfilter.kernelnotes.org/ipchains/spanish/HOWTO.html
