Linux iptables HOWTO

Rusty Russell, mailing list `netfilter@lists.samba.org`

v0.0.2, Wed Sep 29 17:44:43 CST 1999

This document describes how to use iptables to filter out bad packets for Linux kernels 2.3.15 and beyond.

---

1. Introduction

2. Where is the official Web Site and List?

3. What Is Packet Filtering?

4. Who the hell are you, and why are you playing with my kernel?

5. How Packets Traverse The Filters

6. Using iptables

7. Differences Between iptables and ipchains

---

1. Introduction

Welcome, gentle reader.

This HOWTO flips between a gentle introduction (which will leave you feeling warm and fuzzy now, but unprotected in the Real World) and raw full-disclosure (which would leave all but the hardiest souls confused, paranoid and seeking heavy weaponry).

Your network is not **secure**. The problem of allowing rapid, convenient communication while restricting its use to good, and not evil intents is congruent to other intractable problems such as allowing free speech while disallowing a call of ''Fire!'' in a crowded theater. It will not be solved in the space of this HOWTO.

So only you can decide where the compromise will be. I will try to instruct you in the use of some of the tools available and some vulnerabilities to be aware of, in the hope that you will use them for good, and not evil purposes. Another equivalent problem.

2. Where is the official Web Site and List?

There are three official sites:

- Thanks to Penguin Computing.
- Thanks to The Samba Team and SGI.
- Thanks to Jim Pick.

For the official netfilter mailing list, see Samba's Listserver.

3. What Is Packet Filtering?

3.1 Networking (Very) Basics

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.

3.2 So What's A Packet Filter?

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 (as a kernel module at the moment), 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.

3.3 Why Would I Want to Packet Filter?

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.

3.4 How Do I Packet Filter Under Linux?

Linux kernels have had packet filtering since the 1.1 series. The first generation, based on ipfw from BSD, was ported by Alan Cox in late 1994. This was enhanced by Jos Vos and others for Linux 2.0; the userspace tool 'ipfwadm' controlled the kernel filtering rules. In mid-1998, for Linux 2.2, I reworked the kernel quite heavily, with the help of Michael Neuling, and introduced the userspace tool 'ipchains'. Finally, the fourth-generation tool, 'iptables', and another kernel rewrite occurred in mid-1999 for Linux 2.4. It is this iptables which this HOWTO concentrates on.

You need a kernel which has the netfilter infrastructure in it: netfilter is a general framework inside the Linux kernel which other things (such as the iptables module) can plug into. This means you need kernel 2.3.15 or beyond, and answer 'Y' to CONFIG_NETFILTER in the kernel configuration.

The tool `iptables` 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.

iptables

The `iptables` tool inserts and deletes rules from the kernel's packet filtering table. 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.

`iptables` is a replacement for `ipfwadm` and `ipchains`: see Using ipchains and ipfwadm for how to painlessly avoid using iptables if you're using one of those tools.

Making Rules Permanent

Your current firewall setup is stored in the kernel, and thus will be lost on reboot. Writing iptables-save and iptables-restore is on my TODO list. When they exist, they'll be cool, I promise.

Meanwhile, put the command required to set up your rules in an initialization script. Make sure you do something intelligent if one of the commands should fail (usually 'exec /sbin/sulogin').

---

4. Who the hell are you, and why are you playing with my kernel?

I'm Rusty; the Linux IP Firewall maintainer and just another decent coder who happened to be in the right place at the right time. I wrote ipchains (see How Do I Packet Filter Under Linux? above for due credit to the people who did the actual work), and learnt enough to get packet filtering right this time. I hope.

WatchGuard, an excellent firewall company who sell the really nice plug-in Firebox, offered to pay me to do nothing, so I could spend all my time writing this stuff, and maintaining my previous stuff. I predicted 6 months, and it took 12, but I felt by the end that it had been done Right. Many rewrites, a hard-drive crash, a laptop being stolen, a couple of corrupted filesystems and one broken screen later, here it is.
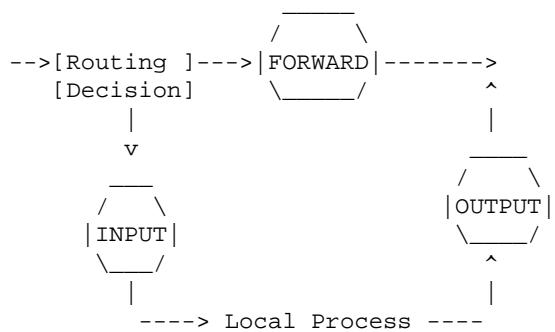
While I'm here, I want to clear up some people's misconceptions: I am no kernel guru. I know this, because my kernel work has brought me into contact with some of them: David S. Miller, Alexey Kuznetsov, Andi Kleen, Alan Cox. However, they're all busy doing the deep magic, leaving me to wade in the shallow end where it's safe.

5. 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**.

This is very different from how the 2.0 and 2.2 kernels worked!

For ASCII-art fans, the chains are arranged like so:

```
                       _____
                      /     \
   -->[Routing ]--->|FORWARD|------->
      [Decision]      \_____/        ^
           |                         |
           v                        ____
          ___                      /    \
         /   \                    |OUTPUT|
        |INPUT|                    \_____/
         \___/                        ^
           |                          |
               ----> Local Process ----
```

The three circles represent the three chains mentioned above. When a packet reaches a circle in the diagram, that chain is examined to decide the fate of the packet. If the chain says to DROP the packet, it is killed there, but if the chain says to ACCEPT the packet, it continues traversing the diagram.

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 DROP the packet.

1. When a packet comes in (say, through the Ethernet card) the kernel first looks at the destination of the packet: this is called 'routing'.
2. If it's destined for this box, the packet passes downwards in the diagram, to the INPUT chain. If it passes this, any processes waiting for that packet will receive it.
3. Otherwise, if the kernel does not have forwarding enabled, or it doesn't know how to forward the packet, the packet is dropped. If forwarding is enabled, and the packet is destined for another network interface (if you have another one), then the packet goes rightwards on our diagram to the FORWARD chain. If it is ACCEPTed, it will be sent out.
4. Finally, a program running on the box can send network packets. These packets pass through the OUTPUT chain immediately: if it says ACCEPT, then the packet continues out to whatever interface it is destined for.

6. Using iptables

iptables has a fairly detailed manual page (`man iptables`), and if you need more detail on particulars. Those of you familiar with ipchains may simply want to look at Differences Between iptables and ipchains; they are very similar.

There are several different things you can do with `iptables`. 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).

6.1 What You'll See When Your Computer Starts Up

At the moment (Linux 2.3.15), iptables is a module, called ('iptables.o'). You'll need to insert it into your kernel before you can use the iptables command. In future, it will be possible to build it in to the kernel.

Before any iptables commands have been run (be careful: some distributions will run iptables in their initialization scripts), there will be no rules in any of the built-in chains ('INPUT', 'FORWARD' and 'OUTPUT'), the INPUT and OUTPUT chains will have a policy of ACCEPT, and the FORWARD chain will have a policy of DROP (you can override this by providing the 'forward=1' option to the iptables module).

6.2 Operations on a Single Rule

This is the bread-and-butter of packet filtering; 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 drop 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 'DROP'.

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
# iptables -A INPUT -s 127.0.0.1 -p icmp -j DROP
# 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 DROP ('-j DROP').

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:

```
# iptables -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 DROP
#
```

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.

6.3 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', '--source' or '--src') and destination ('-d', '--destination' or '--dst') 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.

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 an Interface

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

Packets traversing the INPUT chain don't have an output interface, so any rule using '-o' in this chain will never match. Similarly, packets traversing the OUTPUT chain don't have an input interface, so any rule using '-i' in this chain will never match.

Only packets traversing the FORWARD chain have both an input and output interface.

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 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 these fragments to reconstruct the whole packet.

The problem with fragments is that after the IP header is a part of the internal packet: looking inside the packet for protocol headers (such as is done by the TCP, UDP and ICMP extensions) is not possible, as these headers are only contained in the first fragment.

If you are doing connection tracking or NAT, then all fragments will get merged back together before they reach the packet filtering code, so you need never worry about fragments. Otherwise, you can insert the tiny 'ip_defrag.o' module which performs the same task (note, this is only allowed if you are the only connection between the two networks).

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 --sport www` (specifying a source port of 'www') will never match a fragment (other than the first fragment). Neither will the opposite rule `-p TCP --sport ! www`.

However, you can specify a rule specifically for second and further fragments, using the '-f' (or '--fragment') flag. 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 dropped when such examinations are attempted. So are TCP fragments starting at position 8.

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

```
# iptables -A OUTPUT -f -d 192.168.1.1 -j DROP
#
```

Extensions to iptables: New Tests

`iptables` is **extensible**, meaning that both the kernel and the iptables tool can be extended to provide new features.

Some of these extensions are standard, and other are more exotic. Extensions can be made by other people and distributed separately for niche users.

Kernel extensions normally live in the kernel module subdirectory, such as /lib/modules/2.3.15/net. They are currently (Linux 2.3.15) not demand loaded, so you will need to manually insert the ones you want. In future they may be loaded on-demand again.

Extensions to the iptables program are shared libraries which live usually live in /usr/local/lib/iptables/, although a distribution would put them in /lib/iptables or /usr/lib/iptables.

Extensions come in two types: new targets, and new tests; we'll talk about new targets below. Some protocols automatically offer new tests: currently these are TCP, UDP and ICMP as shown below.

For these you will be able to specify the new tests on the command line after the '-p' option, which will load the extension. For explicit new tests, the '-m' option to load the extension, after which the extended options will be available.

To get help on an extension, use the option to load it ('-p' or '-m') followed by '-h' or '--help'.

TCP Extensions

The TCP extensions are automatically loaded if '--protocol tcp' is specified, and no other match is specified. It provides the following options (none of which match fragments).

--tcp-flags

Followed by an optional '!', then two strings of flags, allows you to filter on specific TCP flags. The first string of flags is the mask: a list of flags you want to examine. The second string of flags tells which one(s) should be set. For example,

```
 # iptables -A INPUT --protocol tcp --tcp-flags ALL SYN,ACK -j DENY
```

This indicates that all flags should be examined ('ALL' is synonymous with 'SYN,ACK,FIN,RST,URG,PSH'), but only SYN and ACK should be set. There is also an argument 'NONE' meaning no flags.

--syn

Optionally preceded by a '!', this is shorthand for '--tcp-flags SYN,RST,ACK SYN'.

--source-port

followed by an optional '!', then either a single TCP port, or a range of ports. Ports can be port names, as listed in /etc/services, or numeric. Ranges are either two port names separated by a '-', or (to specify greater than or equal to a given port) a port with a '-' appended, or (to specify less than or equal to a given port), a port preceded by a '-'.

--sport

is synonymous with '--source-port'.

--destination-port

and

--dport

> are the same as above, only they specify the destination, rather than source, port to match.

--tcp-option

> followed by an optional '!' and a number, matches a packet with a TCP option equaling that number. A packet which does not have a complete TCP header is dropped automatically if at attempt is made to examine its TCP options.

An Explanation of TCP Flags

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 for short). By disallowing only these packets, we can stop attempted connections in their tracks.

The '--syn' 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 --syn
```

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

UDP Extensions

These extensions are automatically loaded if '--protocol udp' is specified, and no other match is specified. It provides the options '--source-port', '--sport', '--destination-port' and '--dport' as detailed for TCP above.

ICMP Extensions

This extension is automatically loaded if '--protocol icmp' is specified, and no other match is specified. It provides only one new option:

--icmp-type

> followed by an optional '!', then either an icmp type name (eg 'host-unreachable'), or a numeric type (eg. '3'), or a numeric type and code separated by a '/' (eg. '3/3'). A list of available icmp type names is given using '-p icmp --help'.

Other Match Extensions

The other two extensions in the netfilter package are demonstration extensions, which (if installed) can be invoked with the '-m' option.

mac

> This module must be explicitly specified with '-m mac' or '--match mac'. It is used for matching incoming packet's source Ethernet (MAC) address, and thus only useful for packets traversing the INPUT and FORWARD chains. It provides only one option:
>
> **--mac-source**
>
> > followed by an optional '!', then an ethernet address in colon-separated hexbyte notation, eg '--mac-source 00:60:08:91:CC:B7'.

limit

> This module must be explicitly specified with '-m limit' or '--match limit'. It is used to restrict the rate of matches, such as for suppressing log messages. It will only match a given number of times per second (by default 3 matches per hour, with a burst of 5). It takes two optional arguments:
>
> **--limit**
>
> > followed by a number; specifies the maximum average number of matches to allow per second. The number can specify units explicitly, using '/second', '/minute', '/hour' or '/day', or parts of them (so '5/second' is the same as '5/s').
>
> **--limit-burst**
>
> > followed by a number, indicating the maximum burst before the above limit kicks in.
>
> This match can often be used with the LOG target to do rate-limited logging. To understand how it works, let's look at the following rule, which logs packets with the default limit parameters:
>
> ```
> # iptables -A FORWARD -m limit -j LOG
> ```
>
> The first time this rule is reached, the packet will be logged; in fact, since the default burst is 5, the first five packets will be logged. After this, it will be twenty minutes before a packet will be logged from this rule, regardless of how many packets reach it. Also, every twenty minutes which passes without matching a packet, one of the burst will be regained; if no packets hit the rule for 100 minutes, the burst will be fully recharged; back where we started.
>
> You cannot currently create a rule with a recharge time greater than about 59 hours, so if you set an average rate of one per day, then your burst rate must be less than 3.

unclean

> This module must be explicitly specified with '-m unclean or '--match unclean'. It does various random sanity checks on packets. This module has not been audited, and should not be used as a security device (it probably makes things worse, since it may well have bugs itself). It provides no options.

6.4 Target Specifications

Now we know what examinations we can do on a packet, we need a way of saying what to do to the packets which match our tests. This is called a rule's **target**.

There are two very simple built-in targets: DROP and ACCEPT. We've already met them. If a rule matches a packet and its target is one of these two, no further rules are consulted: the packet's fate has been decided.
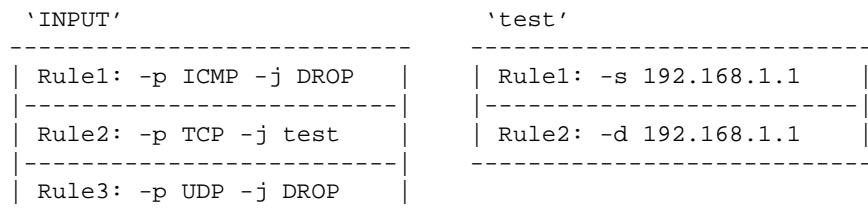
There are two types of targets other than the built-in ones: extensions and user-defined chains.

User-defined chains

One powerful feature which `iptables` inherits from `ipchains` is the ability for the user to create new chains, in addition to the three built-in ones (INPUT, FORWARD and OUTPUT). By convention, user-defined chains are lower-case to distinguish them (we'll describe how to create new user-defined chains below in Operations on an Entire Chain below).
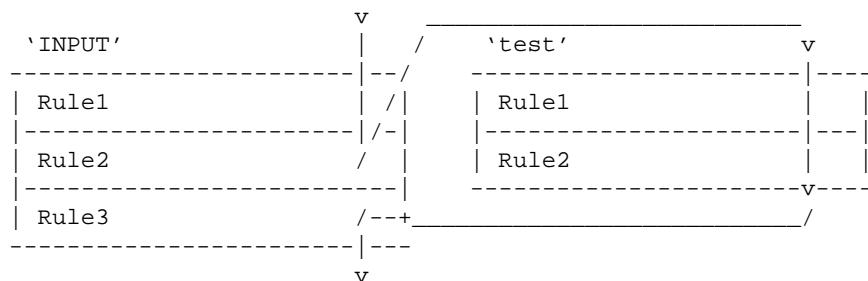
When a packet matches a rule whose target is a user-defined chain, the packet begins traversing the rules in that user-defined chain. If that chain doesn't decide the fate of the packet, then once traversal on that chain has finished, traversal resumes on the next rule in the current chain.

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 DROP  |     | Rule1: -s 192.168.1.1    |
|-------------------------|     |-------------------------|
| Rule2: -p TCP -j test   |     | Rule2: -d 192.168.1.1    |
|-------------------------|     ---------------------------
| Rule3: -p UDP -j DROP   |
---------------------------
```

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
```

15

User-defined chains can jump to other user-defined chains (but don't make loops: you're packets will be dropped if they're found to be in a loop).

Extensions to iptables: New Targets

The other type of target is an extension. A target extension consists of a kernel module, and an optional extension to `iptables` to provide new command line options. There are several extensions in the default netfilter distribution:

LOG

> This module provides kernel logging of matching packets. It provides these additional options:
> **--log-level**
>
> > Followed by a level number or name. Valid names are (case-insensitive) 'debug', 'info', 'notice', 'warning', 'err', 'crit', 'alert' and 'emerg', corresponding to numbers 7 through 0. See the man page for syslog.conf for an explanation of these levels.
>
> **--log-prefix**
>
> > Followed by a string of up to 14 characters, this message is sent at the start of the log message, to allow it to be uniquely identified.
>
> This module is most useful after a limit target, so you don't flood your logs.

REJECT

> This module has the same effect as 'DROP', except that the sender is sent an ICMP 'port unreachable' error message. Note that the ICMP error message is not sent if (see RFC 1122):
>
> - The packet being filtered was an ICMP error message in the first place, or some unknown ICMP type.
> - The packet being filtered was a non-head fragment.
> - We've sent too many ICMP error messages to that destination recently.

Special Built-In Targets

There are two special built-in targets: `RETURN` and `QUEUE`.

`RETURN` has the same effect of falling off the end of a chain: for a rule in a built-in chain, the policy of the chain is executed. For a rule in a user-defined chain, the traversal continues at the previous chain, just after the rule which jumped to this chain.

`QUEUE` is a special target, which queues the packet for userspace processing. Unless there is something waiting for the packet (ie. a program which hasn't been written yet), the packet will be dropped.

6.5 Operations on an Entire Chain

A very useful feature of `iptables` is the ability to group related rules into chains. You can call the chains whatever you want, but I recommend using lower-case letters to avoid confusion with the built-in chains and targets. Chain names can be up to 16 letters long.

Creating a New Chain

Let's create a new chain. Because I am such an imaginative fellow, I'll call it `test`. We use the '-N' or '--new-chain' options:

```
# iptables -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, using the '-X' or '--delete-chain' options. Why '-X'? Well, all the good letters were taken.

```
# iptables -X test
#
```

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.

If you don't specify a chain, then *all* user-defined chains will be deleted, if possible.

Flushing a Chain

There is a simple way of emptying all rules out of a chain, using the '-F' (or '--flush') commands.

```
        # 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.

The 'refcnt' listed for each user-defined chain is the number of rules which have that chain 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 `iptables` 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 TCP and UDP 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 comparisons, and the interfaces. Otherwise these values are omitted.

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' (or '--zero') option.

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.

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 earlier. 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 either ACCEPT or DROP.

Using ipchains and ipfwadm

There are modules in the netfilter distribution called ipchains.o and ipfwadm.o. Insert one of these in your kernel (NOTE: they are incompatible with iptables.o, ip_conntrack.o and ip_nat.o!). Then you can use ipchains or ipfwadm just like the good old days.

This will be supported for some time yet. I think a reasonable formula is 2 * [notice of replacement - initial stable release], beyond the date that a stable release of the replacement is available.

This means that for ipfwadm, the end of support is (FIXME: get real dates):

```
  2 * [October 1997 (2.1.102 release) - March 1995 (ipfwadm 1.0)]
         + January 1999 (2.2.0 release)
    = November 2003.
```

This means that for ipchains, the end of support is (FIXME: get real dates):

```
  2 * [August 1999 (2.3.15 release) - October 1997 (2.2.0 release)]
         + January 2000 (2.3.0 release?)
    = September 2003.
```

So you don't have to worry until 2004.

---

7. Differences Between iptables and ipchains

- Firstly, the names of the built-in chains have changed from lower case to UPPER case, because the INPUT and OUTPUT chains now only get locally-destined and locally-generated packets. They used to see all incoming and all outgoing packets respectively.
- The '-i' flag now means the incoming interface, and only works in the INPUT and FORWARD chains. Rules in the FORWARD or OUTPUT chains that used '-i' should be changed to '-o'.
- TCP and UDP ports now need to be spelled out with the --source-port or --sport (or --destination-port/--dport) options, and must be placed after the '-p tcp' or '-p udp' options, as this loads the TCP or UDP extensions respectively (you may need to insert the ipt_tcp and ipt_udp modules manually).
- The TCP -y flag is now --syn, and must be after '-p tcp'.
- The DENY target is now DROP, finally.
- Zeroing single chains while listing them works.
- Zeroing built-in chains also clears policy counters.
- Listing chains gives you the counters as an atomic snapshot.
- REJECT and LOG are now extended targets, meaning they are separate kernel modules.
- Chain names can be up to 16 characters.
- MASQ and REDIRECT are no longer targets; iptables doesn't do packet mangling. There is a separate NAT subsystem for this: see the ipnatctl HOWTO.
- Probably heaps of other things I forgot.