Linux 2.4 Packet Filtering HOWTO

Rusty Russell, mailing list `netfilter@lists.samba.org` $Revision: 1.21 $ $Date: 2001/08/15 11:03:17 $

This document describes how to use iptables to filter out bad packets for the 2.4 Linux kernels.

Contents

1 Introduction

Welcome, gentle reader.

It is assumed you know what an IP address, a network address, a netmask, routing and DNS are. If not, I recommend that you read the Network Concepts HOWTO.

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? Is there a Mailing List?

There are three official sites:

- Thanks to *Filewatcher* `<http://netfilter.filewatcher.org/>`.
- Thanks to *The Samba Team and SGI* `<http://netfilter.samba.org/>`.
- Thanks to *Harald Welte* `<http://netfilter.gnumonks.org/>`.

For the official netfilter mailing list, see *Samba's Listserver* `<http://lists.samba.org/>`.

3 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 **DROP** the packet (i.e., discard the packet as if it had never received it), **ACCEPT** the packet (i.e., let the packet go through), or something more complicated.

Under Linux, packet filtering is built into the kernel (as a kernel module, or built right in), 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.

3.1 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: see Junkbuster).

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.

3.2 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.

3.2.1 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 3.2.2 (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 8 (Using ipchains and ipfwadm) for how to painlessly avoid using iptables if you're using one of those tools.

3.2.2 Making Rules Permanent

Your current firewall setup is stored in the kernel, and thus will be lost on reboot. You can try the iptables-save and iptables-restore scripts to save them to, and restore them from a file.

The other way is to put the commands 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 Russell; the Linux IP Firewall maintainer and just another working coder who happened to be in the right place at the right time. I wrote ipchains (see 3.2 (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 `<http://www.watchguard.com>`, 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 Rusty's Really Quick Guide To Packet Filtering

Most people just have a single PPP connection to the Internet, and don't want anyone coming back into their network, or the firewall:

```
## Insert connection-tracking modules (not needed if built into kernel).
# insmod ip_conntrack
# insmod ip_conntrack_ftp

## Create chain which blocks new connections, except if coming from inside.
# iptables -N block
# iptables -A block -m state --state ESTABLISHED,RELATED -j ACCEPT
# iptables -A block -m state --state NEW -i ! ppp0 -j ACCEPT
# iptables -A block -j DROP

## Jump to that chain from INPUT and FORWARD chains.
# iptables -A INPUT -j block
# iptables -A FORWARD -j block
```

6 How Packets Traverse The Filters

The kernel starts with three lists of rules in the 'filter' table; these lists are called **firewall chains** or just **chains**. The three chains are called **INPUT**, **OUTPUT** and **FORWARD**.

For ASCII-art fans, the chains are arranged like so: **(Note: this is a very different arrangement from the 2.0 and 2.2 kernels!)**

```
                          -----
Incoming                 /     \         Outgoing
       -->[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.

7 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 10 (Differences Between iptables and ipchains); they are very similar.

There are several different things you can do with `iptables`. You start with three built-in chains `INPUT`, `OUTPUT` and `FORWARD` which you can't delete. Let's look at the operations to manage whole chains:

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).

7.1 What You'll See When Your Computer Starts Up

iptables may be a module, called ('iptable_filter.o'), which should be automatically loaded when you first run `iptables`. It can also be built into the kernel permenantly.

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'), all the chains will have a policy of ACCEPT. You can alter the default policy of the FORWARD chain by providing the 'forward=0' option to the iptable_filter module.

7.2 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:

```
# iptables -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.

7.3 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.

7.3.1 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:

```
[ NOTE: '-s 0/0' is redundant here. ]
# iptables -A INPUT -s 0/0 -j DROP
#
```

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

7.3.2 Specifying Inversion

Many flags, including the '-s' (or '–source') and '-d' ('–destination') 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.

7.3.3 Specifying Protocol

The protocol can be specified with the '-p' (or '–protocol') 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' to specify packets which are **not** TCP.

7.3.4 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' (i.e., 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 '!' with spaces around it, to match a packet which does **not** match the specified interface(s), eg `-i ! ppp+`.

7.3.5 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 the initial fragment has the complete header fields (IP + TCP, UDP and ICMP) to examine, but subsequent packets only have a subset of the headers (IP without the additional protocol fields). Thus looking inside subsequent fragments for protocol headers (such as is done by the TCP, UDP and ICMP extensions) is not possible.

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, 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
#
```

7.3.6 Extensions to iptables: New Matches

`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/2.4.0-test10/kernel/net/ipv4/netfilter. They are demand loaded if your kernel was compiled with CONFIG_KMOD set, so you should not need to manually insert them.

Extensions to the iptables program are shared libraries which 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 matches (we'll talk about new targets a little later). 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, use 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', '-j' or '-m') followed by '-h' or '–help', eg:

```
# iptables -p tcp --help
#
```

TCP Extensions The TCP extensions are automatically loaded if '-p tcp' 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 DROP
```

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 an 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 RST 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 '-p udp' 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 '-p icmp' 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 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 PREROUTING and INPUT 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.

Note: 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.

You can also use this module to avoid various denial of service attacks (DoS) with a faster rate to increase responsiveness.

Syn-flood protection:

```
# iptables -A FORWARD -p tcp --syn -m limit --limit 1/s -j ACCEPT
```

Furtive port scanner:

```
# iptables -A FORWARD -p tcp --tcp-flags SYN,ACK,FIN,RST RST -m limit --limit 1/s -j ACCEPT
```

Ping of death:

```
# iptables -A FORWARD -p icmp --icmp-type echo-request -m limit --limit 1/s -j ACCEPT
```

This module works like a "hysteresis door", as shown in the graph below.

```
        rate (pkt/s)
              ^        .---.
              |       / DoS \
              |      /       \
Edge of DoS  -|.....:.........\.......................
 = (limit *   |    /:          \
limit-burst)  |   / :           \         .-.
              |  /  :            \       /   \
              | /   :             \     /     \
End of DoS   -|/....:..............:.../.......\..../.
 = limit      |     :              :`-'         `--'
-------------+-----+--------------+------------------> time (s)
   LOGIC =>  Match | Didn't Match |    Match
```

Say we say match one packet per second with a five packet burst, but packets start coming in at four per second, for three seconds, then start again in another three seconds.

```
        <--Flood 1-->           <---Flood 2--->

Total  ^                   Line  __--      YNNN
Packets|               Rate  __--      YNNN
       |            mum  __--      YNNN
    10 |        Maxi __--         Y
       |         __--            Y
       |     __--               Y
       | __--    YNNN
       |-    YNNN
     5 |    Y
       |   Y                                Key:  Y -> Matched Rule
       |  Y                                       N -> Didn't Match Rule
       | Y
       |Y
     0 +-------------------------------------------------->  Time (seconds)
        0   1   2   3   4   5   6   7   8   9  10  11  12
```

You can see that the first five packets are allowed to exceed the one packet per second, then the limiting kicks in. If there is a pause, another burst is allowed but not past the maximum rate set by the rule (1 packet per second after the burst is used).

owner

This module attempts to match various characteristics of the packet creator, for locally-generated packets. It is only valid in the OUTPUT chain, and even then some packets (such as ICMP ping responses) may have no owner, and hence never match.

–uid-owner userid

Matches if the packet was created by a process with the given effective (numerical) user id.

–gid-owner groupid

Matches if the packet was created by a process with the given effective (numerical) group id.

–pid-owner processid

Matches if the packet was created by a process with the given process id.

–sid-owner sessionid

Matches if the packet was created by a process in the given session group.

unclean

This experimental 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.

The State Match The most useful match criterion is supplied by the 'state' extension, which interprets the connection-tracking analysis of the 'ip_conntrack' module. This is highly recommended.

Specifying '-m state' allows an additional '–state' option, which is a comma-separated list of states to match (the '!' flag indicates **not** to match those states). These states are:

NEW

A packet which creates a new connection.

ESTABLISHED

A packet which belongs to an existing connection (i.e., a reply packet, or outgoing packet on a connection which has seen replies).

RELATED

A packet which is related to, but not part of, an existing connection, such as an ICMP error, or (with the FTP module inserted), a packet establishing an ftp data connection.

INVALID

A packet which could not be identified for some reason: this includes running out of memory and ICMP errors which don't correspond to any known connection. Generally these packets should be dropped.

7.4 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.

7.4.1 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 7.5 (Operations on an Entire Chain)).

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
```

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

7.4.2 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 29 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 match, 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.

REJECT also takes a '–reject-with' optional argument which alters the reply packet used: see the manual page.

7.4.3 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. For this to be useful, two further components are required:

- a "queue handler", which deals with the actual mechanics of passing packets between the kernel and userspace; and
- a userspace application to receive, possibly manipulate, and issue verdicts on packets.

The standard queue handler for IPv4 iptables is the ip_queue module, which is distributed with the kernel and marked as experimental.

The following is a quick example of how to use iptables to queue packets for userspace processing:

```
# modprobe iptable_filter
# modprobe ip_queue
# iptables -A OUTPUT -p icmp -j QUEUE
```

With this rule, locally generated outgoing ICMP packets (as created with, say, ping) are passed to the ip_queue module, which then attempts to deliver the packets to a userspace application. If no userspace application is waiting, the packets are dropped.

To write a userspace application, use the libipq API. This is distributed with iptables. Example code may be found in the testsuite tools (e.g. redirect.c) in CVS.

The status of ip_queue may be checked via:

```
/proc/net/ip_queue
```

The maximum length of the queue (i.e. the number packets delivered to userspace with no verdict issued back) may be controlled via:

```
/proc/sys/net/ipv4/ip_queue_maxlen
```

The default value for the maximum queue length is 1024. Once this limit is reached, new packets will be dropped until the length of the queue falls below the limit again. Nice protocols such as TCP interpret dropped packets as congestion, and will hopefully back off when the queue fills up. However, it may take some experimenting to determine an ideal maximum queue length for a given situation if the default value is too small.

7.5 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 31 letters long.

7.5.1 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.

7.5.2 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 7.5.3 (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.

7.5.3 Flushing a Chain

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

```
# iptables -F FORWARD
#
```

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

7.5.4 Listing a Chain

You can list all the rules in a chain by using the '-L' (or '–list') 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.

7.5.5 Resetting (Zeroing) Counters

It is useful to be able to reset the counters. This can be done with the '-Z' (or '–zero') option.

Consider the following:

```
# iptables -L FORWARD
# iptables -Z FORWARD
#
```

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.

7.5.6 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`, for example:

```
# iptables -P FORWARD DROP
#
```

8 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 ip_tables.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 support will probably be dropped in Linux 2.6 or 2.8.

9 Mixing NAT and Packet Filtering

It's common to want to do Network Address Translation (see the NAT HOWTO) and packet filtering. The good news is that they mix extremely well.

You design your packet filtering completely ignoring any NAT you are doing. The sources and destinations seen by the packet filter will be the 'real' sources and destinations. For example, if you are doing DNAT to send any connections to 1.2.3.4 port 80 through to 10.1.1.1 port 8080, the packet filter would see packets going to 10.1.1.1 port 8080 (the real destination), not 1.2.3.4 port 80. Similarly, you can ignore masquerading: packets will seem to come from their real internal IP addresses (say 10.1.1.1), and replies will seem to go back there.

You can use the 'state' match extension without making the packet filter do any extra work, since NAT requires connection tracking anyway. To enhance the simple masquerading example in the NAT HOWTO to disallow any new connections from coming in the ppp0 interface, you would do this:

```
# Masquerade out ppp0
iptables -t nat -A POSTROUTING -o ppp0 -j MASQUERADE

# Disallow NEW and INVALID incoming or forwarded packets from ppp0.
iptables -A INPUT -i ppp0 -m state --state NEW,INVALID -j DROP
iptables -A FORWARD -i ppp0 -m state --state NEW,INVALID -j DROP

# Turn on IP forwarding
echo 1 > /proc/sys/net/ipv4/ip_forward
```

10 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.
- 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 31 characters.
- MASQ is now MASQUERADE and uses a different syntax. REDIRECT, while keeping the same name, has also undergone a syntax change. See the NAT-HOWTO for more information on how to configure both of these.
- The -o option is no longer used to direct packets to the userspace device (see -i above). Packets are now sent to userspace via the QUEUE target.
- Probably heaps of other things I forgot.

11 Advice on Packet Filter Design

Common wisdom in the computer security arena is to block everything, then open up holes as neccessary. This is usually phrased 'that which is not explicitly allowed is prohibited'. I recommend this approach if security is your maximal concern.

Do not run any services you do not need to, even if you think you have blocked access to them.

If you are creating a dedicated firewall, start by running nothing, and blocking all packets, then add services and let packets through as required.

I recommend security in depth: combine tcp-wrappers (for connections to the packet filter itself), proxies (for connections passing through the packet filter), route verification and packet filtering. Route verification is where a packet which comes from an unexpected interface is dropped: for example, if your internal network has addresses 10.1.1.0/24, and a packet with that source address comes in your external interface, it will be dropped. This can be enabled for one interface (ppp0) like so:

```
# echo 1 > /proc/sys/net/ipv4/conf/ppp0/rp_filter
#
```

Or for all existing and future interfaces like this:

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

Debian does this by default where possible. If you have asymmetric routing (ie. you expect packets coming in from strange directions), you will want to disable this filtering on those interfaces.

Logging is useful when setting up a firewall if something isn't working, but on a production firewall, always combine it with the 'limit' match, to prevent someone from flooding your logs.

I highly recommend connection tracking for secure systems: it introduces some overhead, as all connections are tracked, but is very useful for controlling access to your networks. You may need to load the 'ip_conntrack.o' module if your kernel does not load modules automatically, and it's not built into the kernel. If you want to accurately track complex protocols, you'll need to load the appropriate helper module (eg. 'ip_conntrack_ftp.o').

```
# iptables -N no-conns-from-ppp0
# iptables -A no-conns-from-ppp0 -m state --state ESTABLISHED,RELATED -j ACCEPT
# iptables -A no-conns-from-ppp0 -m state --state NEW -i ! ppp0 -j ACCEPT
# iptables -A no-conns-from-ppp0 -i ppp0 -m limit -j LOG --log-prefix "Bad packet from ppp0:"
# iptables -A no-conns-from-ppp0 -i ! ppp0 -m limit -j LOG --log-prefix "Bad packet not from ppp0:"
# iptables -A no-conns-from-ppp0 -j DROP

# iptables -A INPUT -j no-conns-from-ppp0
# iptables -A FORWARD -j no-conns-from-ppp0
```

Building a good firewall is beyond the scope of this HOWTO, but my advice is 'always be minimalist'. See the Security HOWTO for more information on testing and probing your box.