IPTables

Project Report

Spring 2011
IIIT, Hyderabad

Sankalp Khare (200702039)

Project Overview

This project documents the configuration and capabilities of the iptables firewall for Linux systems. It also discusses some of the advanced features of iptables, and gives sufficient documentation on how to implement these.

We also describe the firewall config of various servers at IIIT, Hyderabad with suggestions for optimization and overall improvement.

Contents

1 What is IPTables **2**

2 Fundamental Concepts **2**
- 2.1 Chains . . . 2
- 2.2 Tables . . . 3
  - 2.2.1 NAT . . . 3
  - 2.2.2 Filtering . . . 3
  - 2.2.3 Mangling . . . 4
- 2.3 Packet Flow . . . 5
- 2.4 Rules . . . 5
- 2.5 Matches . . . 6
- 2.6 Targets . . . 6

3 Configuring and Managing iptables **6**
- 3.1 Starting and Stopping . . . 6
- 3.2 IPTables List Command . . . 7
- 3.3 IP Tables Flush Command . . . 8
- 3.4 IP Tables Policy Command . . . 9
- 3.5 Writing Rules . . . 9
  - 3.5.1 Leveraging the connection state . . . 10
  - 3.5.2 Blocking packets from specific sources . . . 10
  - 3.5.3 Accepting packets of desired type . . . 11
  - 3.5.4 Saving/Restoring Rules to/from files . . . 11
  - 3.5.5 Custom (User-defined) Chains . . . 11
  - 3.5.6 Logging . . . 13
  - 3.5.7 IPTables matches . . . 14
  - 3.5.8 Implicit matches . . . 15
  - 3.5.9 Explicit matches . . . 18
- 3.6 Some recommended Best Practices for iptables . . . 22

4 Applications **23**

5 Some specific techniques and their configuration **24**
- 5.1 Port Knocking . . . 24
  - 5.1.1 Implementation using custom-chains . . . 24
  - 5.1.2 Using the Portknock0 project iptables module . . . 25
- 5.2 Rate Limiting . . . 25
  - 5.2.1 Protecting against ping flood attacks . . . 26
- 5.3 Connection Limiting . . . 26
  - 5.3.1 Limiting the number of SSH connections from a host (in parallel) . . . 26
  - 5.3.2 Bruteforce attack protection . . . 26

6 iptables at IIIT-H **26**
- 6.1 Suggestions for Optimization and Improvement . . . 27
  - 6.1.1 Optimizing iptables by creating user-defined chains . . . 27
  - 6.1.2 Reordering rules based on counters . . . 27
  - 6.1.3 Logging and Dropping/Accepting with a single rule . . . 28

7 Man Pages **29**

1 What is IPTables

iptables, in the most basic sense, is a firewall program. However, it is one of the most popularly used firewall applications worldwide, and ships with most distributions of Linux. The Linux kernel's network packet processing subsystem is called Netfilter, and iptables is the command used to configure it.

The iptables architecture groups network packet processing rules into tables by function (packet filtering, network address translation, and other packet mangling), each of which have chains (sequences) of processing rules. Rules consist of matches, which are criterions used to determine which packets the rule will apply to, and targets (that determine what will be done with the matching packets). iptables operates at OSI Layer 3 (Network). For OSI Layer 2 (Link), there are other technologies such as ebtables (Ethernet Bridge Tables).

2 Fundamental Concepts

iptables is a stateful firewall. It supports dynamically loadable modules which supplement its workings and provide for a host of extra features. Most of the processing of packets in iptables happens based on chains and tables. The choice of where (table, chain) to put rules is made based on where in the packet's journey we wish to apply those rules, as will become clear in the rest of this section.

2.1 Chains

iptables defines five "hook points" in the kernel's packet processing pathways:

1. PREROUTING
2. INPUT
3. FORWARD
4. POSTROUTING
5. OUTPUT

Built-in chains are attached to these hook points; we can add a sequence of rules for each hook point. Each rule represents an opportunity to affect or monitor packet flow. Saying "INPUT chain" means we are referring to the chain attached to the INPUT hook point.

Each of the "hook points"/chains allows us to manipulate packets at a certain point in their journey through the system :

PREROUTING
Allows us to act on packets just after they arrive, but before any routing decision for them is made.

INPUT
Allows us to process packets just before they are delivered to a local process.

FORWARD
Allows us to process packets that flow through our machine treating it as a gateway, i.e. coming in via one interface and straightaway leaving through another.

POSTROUTING
Allows us to act on packets just before they leave the machine through a network interface.

OUTPUT
Allows us to process packets just after they are generated (by a local process).

In addition, we can create our own custom chains, for better organization of rules.

A chain's policy determines the fate of packets that reach the end of the chain without otherwise being sent to a specific target. Only the built-in targets ACCEPT and DROP can be used as the policy for a built-in chain, and the default is ACCEPT. All user-defined chains have an implicit policy of RETURN that cannot be changed.

It is for this reason that if we want a more complicated policy for a built-in chain or a policy other than RETURN for a user-defined chain, we add a rule to the end of the chain that matches all packets, with a target of our liking.

2.2 Tables

iptables comes with three built-in tables: filter, mangle, and nat. Each is preconfigured with chains corresponding to one or more of the hook points described earlier.

Just as chains represent the hook-points in the iptables workflow, tables represent the type of processing (conceptually) that can occur. The following are the possible legal combinations, and the corresponding tables.

2.2.1 NAT

NAT-ing is the process of modifying the IP Headers, in particular the to/from addresses, of a packet in transit through the machine. A machine performing NAT acts like a routing device.

The NAT Table is used with connection tracking to redirect connections for network address translation; typically based on source or destination addresses.

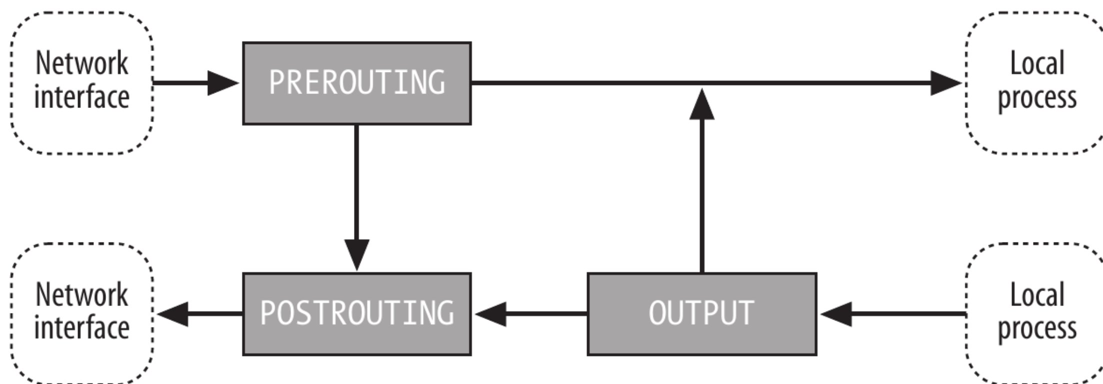

Figure 1: NAT: packet flow and hook points

Figure 1 shows how packets go through the system for Network Address Translation (NAT). It becomes clear that the NAT Table should contain 3 chains – PREROUTING, POSTROUTING and OUTPUT.

2.2.2 Filtering

Filtering is the most widely used feature of iptables. It is used to set policies for the type of traffic allowed into, through, and out of the computer. Unless we refer to a different table explicitly, iptables operates on chains within this table by default. Its built-in chains are: FORWARD, INPUT, and OUTPUT.

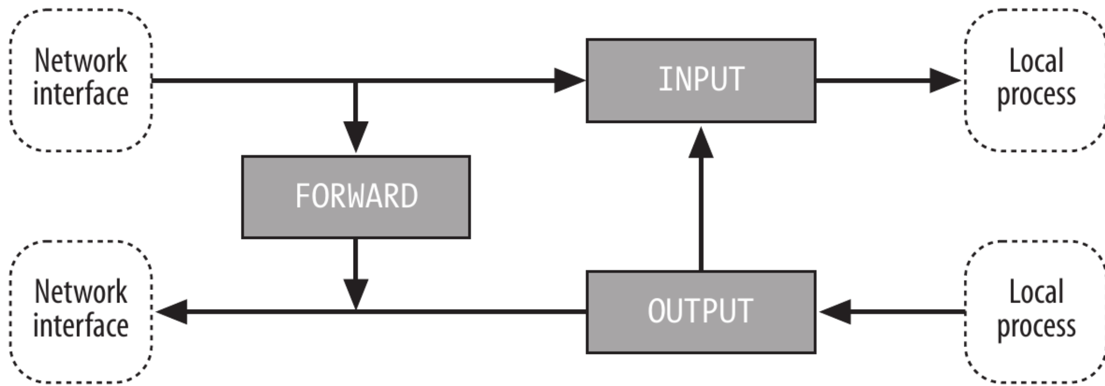

Figure 2: Filtering: packet flow and hook points

Figure 2 shows the (conceptual) flow of packets when they are filtered by the machine. The chains (in the grey boxes) are the chains that become part of the filter table.

2.2.3 Mangling

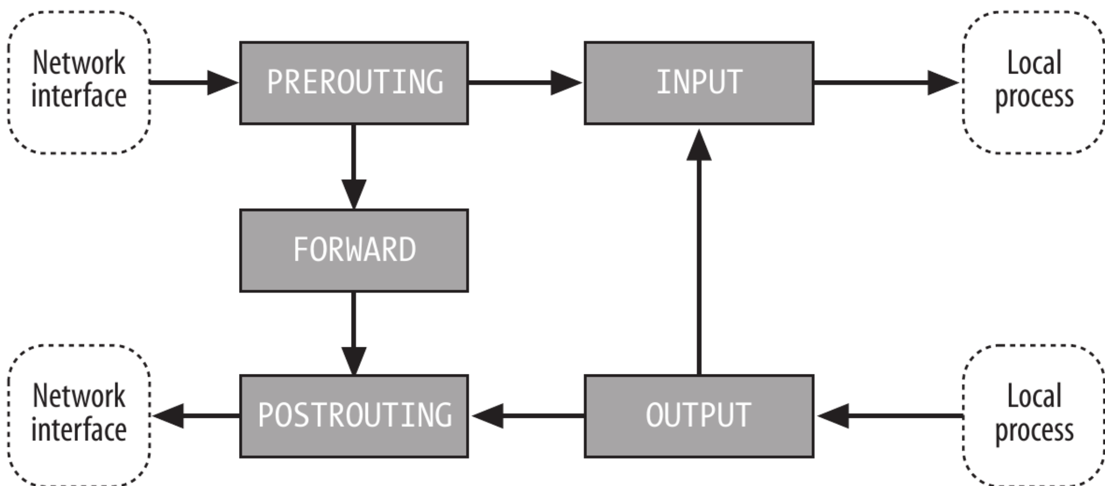

Figure 3: Mangling: packet flow and hook points

Mangling is used in specialized packet alteration, such as stripping off IP options (as with the IPV4OPTSSTRIP target extension). Its built-in chains are: FORWARD, INPUT, OUTPUT, POSTROUTING, and PREROUTING.

Figure 3 shows the corresponding flow for packet mangling. We will not discuss much of mangling in this document.

Thus, if all the above were to be put into one grand schematic, it would resemble figure 4

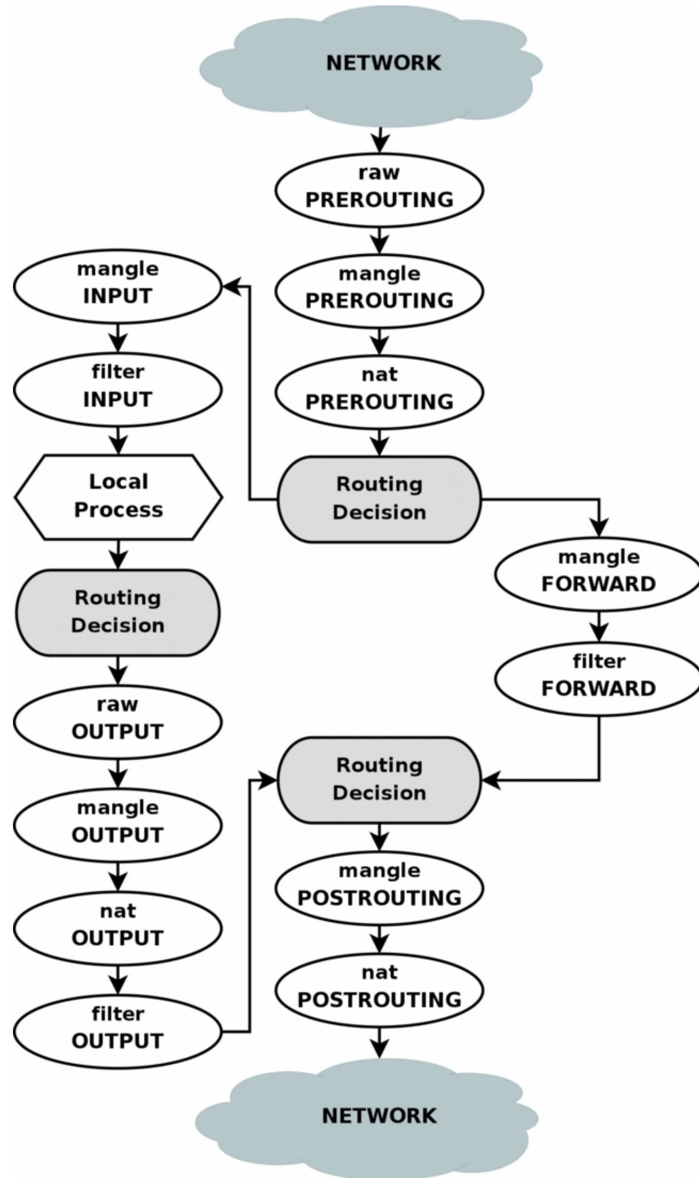

Figure 4: IPTables: The grand scheme of things

2.3 Packet Flow

Packets traverse chains, and are presented to their rules one at a time in order. If a packet does not match the rule's criteria, the packet moves to the next rule in the chain. If a packet reaches the last rule in the chain and still does not match, the chain's policy (which can also be viewed as the chain's default target) is applied to it.

2.4 Rules

An iptables rule consists of one or more match criteria that determine which network packets it affects (all match options must be satisfied for the rule to match a packet) and a target specification that determines how the network packets will be affected. The system maintains packet and byte counters for every rule. Every time a packet reaches a rule and matches the rule's criteria, the packet counter is incremented, and the byte counter is increased by the size of the matching packet.

Both the match and the target portion of the rule are optional. If there are no match criteria, all packets are considered to match. If there is no target specification, nothing is done to the packets (processing proceeds as if the rule did not exist—except that the packet and byte counters are updated).

2.5 Matches

There are a variety of matches available for use with iptables, although some are available only for kernels with certain features enabled (usually later versions of kernels). Generic Internet Protocol (IP) matches (such as protocol, source, or destination address) are applicable to any IP packet. In addition to the generic matches, iptables includes many specialized matches available through dynamically loaded extensions (we use the iptables -m or --match option to inform iptables that we want to use one of these extensions). There is one match extension for dealing with a networking layer below the IP layer. The mac match extension matches based on Ethernet media access controller (MAC) addresses.

2.6 Targets

Targets are used to specify the action to take when a rule matches a packet and also to specify chain policies. Four targets are built into iptables, and extension modules can provide more.

The built in targets are :

ACCEPT
Let the packet through to the next stage of processing. Stop traversing the current chain, and start at the next stage in the packet flow.

DROP
Discontinue processing the packet completely. Do not check it against any other rules, chains, or tables. In case we want to provide some feedback to the sender, we must use the REJECT target instead.

QUEUE
Send the packet to userspace (i.e. code not in the kernel). The libipq manpage offers more information.

RETURN
From a rule in a user-defined chain, discontinue processing this chain, and resume traversing the calling chain at the rule following the one that had this chain as its target. From a rule in a built-in chain, discontinue processing the packet and apply the chain's policy to it.

3 Configuring and Managing iptables

3.1 Starting and Stopping

iptables can be started/stopped using the `service` command.

Listing 1: Starting and Stopping iptables

```
1  [root@someserver ~]$ service iptables start
2  [root@someserver ~]$ service iptables stop
```

Whether or not to invoke it on startup can be set using the `chkconfig` command.

When iptables is started, the default rule-set is loaded from the contents of the file `/etc/sysconfig/iptables`. This file should contain a list of iptables rules that we want to

add, in the same order. The only difference is that while writing them in this file, we can omit the `iptables` prefix.

Listing 2: Sample /etc/sysconfig/iptables file

```
# Firewall configuration written by system-config-firewall
# Manual customization of this file is not recommended.
*filter
:INPUT ACCEPT [0:0]
:FORWARD ACCEPT [0:0]
:OUTPUT ACCEPT [0:0]
-A INPUT -m state --state ESTABLISHED,RELATED -j ACCEPT
-A INPUT -p icmp -j ACCEPT
-A INPUT -i lo -j ACCEPT
-A INPUT -m state --state NEW -m tcp -p tcp --dport 22 -j ACCEPT
-A INPUT -j REJECT --reject-with icmp-host-prohibited
-A FORWARD -j REJECT --reject-with icmp-host-prohibited
COMMIT
```

In listing 2,

- line 3 means that the subsequent rules are part of the FILTER table.
- lines 4, 5 and 6 set the default policy of the INPUT, FORWARD and OUTPUT chains to ACCEPT, and the counters to zero.
- lines 7-12 (inclusive) are the desired rules to be loaded on startup.
- line 13 enforces the above configuration.

note: now onwards, we may describe rules without explicitly prefixing them with the `iptables` *command.*

3.2 IPTables List Command

The iptables list command displays all the chains, with their rules, that are currently in place.

Listing 3: Listing config: Empty configuration

```
[root@someserver ~]$ iptables -L
Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

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

The above output indicates that there are 3 chains, but all of them are empty. This is also the output shown when iptables has been switched off.

A simple configuration may be like the following:

Listing 4: Listing config: Basic configuration

```
[root@someserver ~]$ iptables -L
Chain INPUT (policy ACCEPT)
target     prot opt source       destination
ACCEPT     all  --  anywhere     anywhere       state RELATED,ESTABLISHED
ACCEPT     icmp --  anywhere     anywhere
ACCEPT     all  --  anywhere     anywhere
ACCEPT     tcp  --  anywhere     anywhere       state NEW tcp dpt:ssh
REJECT     all  --  anywhere     anywhere       reject-with icmp-host-prohibited

```

```
Chain FORWARD (policy ACCEPT)
target     prot opt source       destination
REJECT     all  --  anywhere     anywhere      reject-with icmp-host-prohibited

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

Each IP Tables command has two versions a shorthand version and a verbose version. Following the general style used by most linux command line utilities, the verbose versions all use a double dash and the shorthand versions use a single dash.

Listing 5: Listing config: Some other invocations

```
[root@someserver ~]$ iptables --list            # verbose version
[root@someserver ~]$ iptables -L --line-numbers # listing with line numbers
[root@someserver ~]$ iptables -L -n             # list ports and IP addresses
    numerically, rather than by name
```

Listing ports and IP addresses numerically (as shown in the last example of listing 5) is highly advisable for quick outputs when the ruleset is large. The reason is that when doing a normal listing, `iptables` performs DNS lookups for the addresses in each rule, which may cause a considerable delay in printing the output.

Listing 6: Difference in output of normal and numeric listing

```
[root@someserver ~]$ iptables -L # normal
Chain INPUT (policy ACCEPT)
target     prot opt source       destination
ACCEPT     all  --  anywhere     anywhere     state RELATED,ESTABLISHED
ACCEPT     icmp --  anywhere     anywhere
ACCEPT     all  --  anywhere     anywhere
ACCEPT     tcp  --  anywhere     anywhere     state NEW tcp dpt:ssh
REJECT     all  --  anywhere     anywhere     reject-with icmp-host-prohibited

Chain FORWARD (policy ACCEPT)
target     prot opt source       destination
REJECT     all  --  anywhere     anywhere     reject-with icmp-host-prohibited

Chain OUTPUT (policy ACCEPT)
target     prot opt source       destination

[root@someserver ~]$ iptables -L -n # numeric
Chain INPUT (policy ACCEPT)
target     prot opt source       destination
ACCEPT     all  --  0.0.0.0/0    0.0.0.0/0    state RELATED,ESTABLISHED
ACCEPT     icmp --  0.0.0.0/0    0.0.0.0/0
ACCEPT     all  --  0.0.0.0/0    0.0.0.0/0
ACCEPT     tcp  --  0.0.0.0/0    0.0.0.0/0    state NEW tcp dpt:22
REJECT     all  --  0.0.0.0/0    0.0.0.0/0    reject-with icmp-host-prohibited

Chain FORWARD (policy ACCEPT)
target     prot opt source       destination
REJECT     all  --  0.0.0.0/0    0.0.0.0/0    reject-with icmp-host-prohibited

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

Notice how, in the numeric version, keywords get replaced with their numeric meanings.

3.3 IP Tables Flush Command

If there are any rules listed we first need to clear them. To do that we use the flush command:

Listing 7: Flush command

```
1  [root@someserver ~]$ iptables --flush
2  [root@someserver ~]$ iptables -F       # shorthand version
```

Usually, while applying any fresh configuration, the first step is to purge the existing one using the flush command.

3.4 IP Tables Policy Command

Using this command, we can set the default policy for each of the chains. As mentioned earlier, the policy is the action to be performed on a packet which passes through the chain without being matched by any rule.

Listing 8: Setting DROP as the policy for all 3 built-in chains

```
1  [root@someserver ~]$ iptables --policy INPUT DROP   # policy for chain INPUT set to
       DROP
2  [root@someserver ~]$ iptables --policy FORWARD DROP # likewise for chain FORWARD
3  [root@someserver ~]$ iptables --policy OUTPUT DROP  # and for chain OUTPUT
```

This sets our overall policy to drop every packet that we don't explicitly allow.

The shorthand version of the policy command is:

Listing 9: Short version

```
1  [root@someserver ~]$ iptables -P INPUT DROP # set the policy for chain INPUT as DROP
```

3.5 Writing Rules

The basic format of an iptables rule is as decribed in listing 10.

Listing 10: iptables rule format

```
1 iptables [-t table] command [match] [target/jump]
```

There is nothing that says that the target instruction has to be the last function in the line. However, we generally adhere to this syntax to get the best readability. Most people write their rules in this way. Hence, if we read someone else's script, it becomes easier to recognize the syntax and easily understand the rule.

If we want to use a table other than the standard table, we insert the table specification at the point at which `[table]` is specified. However, it is not necessary to state explicitly what table to use, since by default iptables uses the filter table on which to implement all commands. Neither do we have to specify the table at just this point in the rule. It could be set pretty much anywhere along the line. However, it is more or less standard to put the table specification at the beginning.

The command always comes first, unless a table is explicitly specified, in which case the table declaration comes first, followed by the command. We use 'command' to tell the program what to do, for example to insert a rule (`-I`) or to add a rule to the end of the chain (`-A`), or to delete a rule (`-D`).

The match is the part of the rule that iptables sends to the kernel that details the specific character of the packet, what makes it different from all other packets. Here we could specify what IP address the packet comes from, from which network interface, the intended IP address, port, protocol or whatever. There is a lot of different matches that we can use.

Finally we have the target of the packet. If all the matches are met for a packet, we tell iptables what to do with it. We could, for example, tell it to send the packet to another chain that we've created ourselves, and which is part of this particular table. We could instruct iptables to drop the packet and do no further processing, or we could make it send a specified reply to the sender.

Lets take a sample iptables rule and find out what each part means.

Listing 11: Sample iptables rule

```
iptables \\         # command
-A INPUT \\         # append this rule to chain INPUT
-s 192.168.1.10 \\ # packet source 192.168.1.10
-d 10.1.15.1 \\     # packet destination 10.1.15.1
-p tcp \\           # protocol TCP
--dport 22 \\       # destination port 22 (SSH)
-j ACCEPT \\        # if everything matches, jump to ACCEPT (action)
```

A general iptables rule lists out a set of conditions and an action such that if a packet matches all the given conditions, the said action should be performed on it.

3.5.1 Leveraging the connection state

iptables can determine the "state" of a TCP packet. This is a feature commonly used to identify packets while writing rules. For example, there is the three-way handshake between two hosts when transmitting data:

1. NEW => server1 connects to server2 issuing a SYN (Synchronize) packet.
2. RELATED => server2 receives the SYN packet, and then responds with a SYN-ACK (Synchronize Acknowledgment) packet.
3. ESTABLISHED => server1 receives the SYN-ACK packet and then responds with the final ACK (Acknowledgment) packet.

The three rules in listing 12, together, allow this kind of TCP Communication.

Listing 12: iptables rules allowing for three-way TCP handshakes

```
-A INPUT \\    # append to chain INPUT
-m state \\    # load module "state"
--state RELATED,ESTABLISHED \\ # packets with state RELATED or ESTABLISHED
-j ACCEPT      # jump to ACCEPT

-A FORWARD \\ # append to chain FORWARD
-i eth0 \\     # packets being forwarded by interface eth0
-m state \\    # load module "state"
--state RELATED,ESTABLISHED \\ # packets with state RELATED or ESTABLISHED
-j ACCEPT      # jump to ACCEPT

-A OUTPUT \\   # append to chain OUTPUT
-m state \\    # load module "state"
--state NEW,RELATED,ESTABLISHED \\ # packets with state NEW, RELATED or ESTABLISHED
-j ACCEPT      # jump to ACCEPT
```

The rules in listing 12 utilize the `state` module of iptables. Note that modules are loaded dynamically, as per requirement, using the `-m <modulename>` switch.

note: the `state` module is now obsoleted by the `conntrack` module, which has enhanced functionality.

3.5.2 Blocking packets from specific sources

The rule in listing 13 blocks all incoming traffic from the IP 10.1.34.246.

Listing 13: iptables rule restricting all incoming traffic from 10.1.34.246

```
iptables \\
-A INPUT \\       # append rule to chain INPUT
-s 10.1.34.246 \\ # packets from source 10.1.34.246
-j DROP           # DROP (regardless of protocol/packet type)
```

The rule in listing 14 blocks all incoming traffic on port 25 (SMTP) from the host mail.spammer.org (assuming it is a host which sends spam mail).

Listing 14: iptables rule preventing mail-spammming from a known spam source

```
iptables \\
-A INPUT \\            # add as the last rule of chain INPUT (append)
-s mail.spammer.org \\ # packet source mail.spammer.org
-p tcp \\              # protocol tcp
--dport 25 \\          # packet destination port 25
-j REJECT              # jump to action REJECT
```

Notice, as shown in listing 14, line 3, that iptables accepts domain names in source/destination specifications.

3.5.3 Accepting packets of desired type

Knowing the port being used for an application allows us to leverage that fact to accept/reject traffic pertaining to that application. The rules in listing 15 do exactly that.

Listing 15: iptables rules permitting traffic of certain types

```
iptables -A INPUT -p tcp --dport 22 -j ACCEPT    # ssh
iptables -A INPUT -p tcp --dport 25 -j ACCEPT    # sendmail/smtp
iptables -A INPUT -p tcp --dport 20:21 -j ACCEPT # ftp
iptables -A INPUT -p tcp --dport 80 -j ACCEPT    # http
iptables -A INPUT -p icmp -j ACCEPT              # icmp/ping packets
```

Notice how, in rule 3 of listing 15, we can specify a range of ports.

3.5.4 Saving/Restoring Rules to/from files

The `iptables-save` and `iptables-restore` commands allow us to save and restore rules to/from files. Consult the man-pages attached, for reference.

note: using iptables-save to redirect output will overwrite `/etc/sysconfig/iptables`, so it must be used with care. Any custom comments etc. present in the file will be lost.

3.5.5 Custom (User-defined) Chains

We can create our own chains, in iptables. This is very helpful in both organizing our firewall config and streamlining it to make it more efficient.

Let us demonstrate the chain management commands by an example :

- Creating a chain

Listing 16: Chain creation

```
iptables --new-chain OUTPUTDROP      # create a chain by name OUTPUTDROP
#  iptables -N OUTPUTDROP            # abbreviated version
iptables --delete-chain OUTPUTDROP   # oops! wrong name... DELETE!
#  iptables -X OUTPUTDROP            # abbreviated version of delete
iptables -N OUTACCEPT                # re-create with the intended name
```

Now we have a chain called OUTACCEPT, which is empty.

- Adding rules to the chain
  Next, we will add rules to the chain. Let us add the exact same rules which are present in the output chain (the reason will become clear soon).

Listing 17: Adding Rules to the Chain

```
iptables -A OUTACCEPT \\ # append to chain OUTACCEPT
         -o lo \\        # packets going out through the loopback interface
         -j ACCEPT       # allow them to pass
iptables -A OUTACCEPT \\
         -o eth0 \\      # packets exiting through interface eth0
         -j ACCEPT
```

Once these rules are added, a rule listing should show output as shown in listing 18

Listing 18: The result

```
[root@someserver ~]$ iptables -L -v --line-numbers
...
...
Chain OUTPUT (policy DROP 0 packets, 0 bytes)
num  pkts bytes target  prot opt in     out     source     destination
1      34  2252 ACCEPT  all  --  any    lo      anywhere   anywhere
2    1651  299K ACCEPT  all  --  any    eth0    anywhere   anywhere

Chain OUTACCEPT (0 references)
num  pkts bytes target  prot opt in     out     source     destination
1       0     0 ACCEPT  all  --  any    lo      anywhere   anywhere
2       0     0 ACCEPT  all  --  any    eth0    anywhere   anywhere
...
...
```

Notice that the chain OUTACCEPT has 0 references, which means there is no rule that points to it, and thus the chain is currently unused.

- Referencing the chain
  Our chain now has the exact same rules as the chain OUTPUT. Let us now modify the OUTPUT chain such that it only contains a single rule which points to chain OUTACCEPT. The procedure is outlined in listing

Listing 19: Referencing our Chain

```
[root@someserver ~]$ iptables \\
                     -I OUTPUT 1 \\ # insert in chain OUTPUT at position 1
                     -j OUTACCEPT   # jump to chain OUTACCEPT
[root@someserver ~]$ iptables -L -v --line-numbers OUTPUT
Chain OUTPUT (policy DROP 0 packets, 0 bytes)
num  pkts bytes target    prot opt in   out   source     destination
1     421 71264 OUTACCEPT all  --  any  any   anywhere   anywhere
2      34  2252 ACCEPT    all  --  any  lo    anywhere   anywhere
3    1651  299K ACCEPT    all  --  any  eth0  anywhere   anywhere
[root@someserver ~]$ iptables -D OUTPUT 2  # delete rule 2 of chain OUTPUT
[root@someserver ~]$ iptables -D OUTPUT 2  # delete rule 2 of chain OUTPUT
```

Notice that in line 11 we again delete rule 2 of chain OUTPUT. This makes sense because after the delete performed in the previous line, only 2 rules remain (the 2nd one being the one which was earlier 3rd).

- The net result
  After all the steps outlined above, the iptables listing of chains OUTPUT and OUTACCEPT should look like what is shown in listing 20.

Listing 20: The final result

```
1   Chain OUTPUT (policy DROP 0 packets, 0 bytes)
2   num  pkts bytes target     prot opt in   out   source    destination
3   1     421 71264 OUTACCEPT all  --  any  any   anywhere  anywhere
4
5   Chain OUTACCEPT (1 references)
6   num  pkts bytes target     prot opt in   out   source    destination
7   1      14   980 ACCEPT     all  --  any  lo    anywhere  anywhere
8   2     407 70284 ACCEPT     all  --  any  eth0  anywhere  anywhere
```

This setup means that any packet going out of the machine will be sent to chain OUTACCEPT, which only matches packets exiting through either `eth0` or `lo` interfaces.
Packets which do not match will then return to chain OUTPUT and since there are no more rules to check, they will get processed according to the chain policy of chain OUTPUT.

It is important to know the behaviour of custom chains. Let's assume a rule, say rule number n of some chain X references a custom chain, Y. When a packet matches that rule, and is passed to chain Y, iptables checks it against all the rules in Y, one by one. If some rule matches, it gets applied. If none of the rules match, then the control is sent back to the parent chain, X, where execution continues from rule $n+1$. If n was the last rule of chain X, then the chain policy of chain X gets applied to the packet.

The flow can be understood from figure 5.

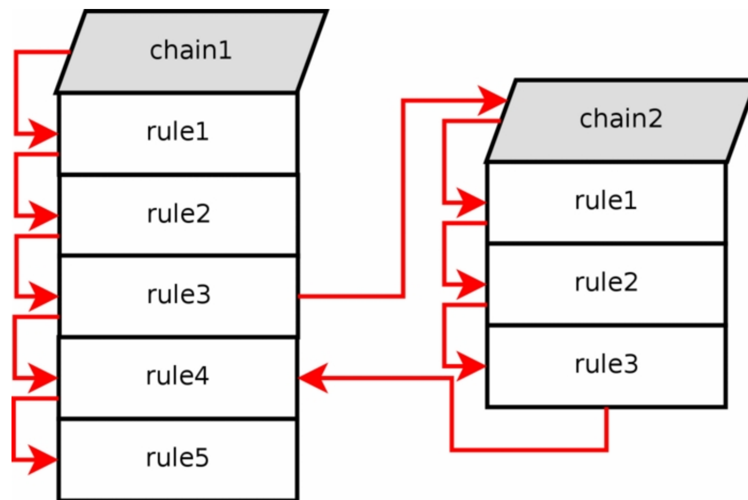

Figure 5: The flow of control in case of custom-chains

3.5.6 Logging

iptables allows us to log matches, by default to the file `/var/log/messages`. Logging is done by specifying the jump target of a rule as LOG. This instructs iptables to write log entries for packets matching that rule.

More often than not, one wishes to do more than just logging, for the packets that match a rule. The standard method is to write two identical rules (in terms of match criteria), but give them different jump targets. Consider the rules in listing 21.

Listing 21: iptables rules to log and drop packets

```
1   -A INPUT -s 10.1.34.40 -j LOG  # make a log entry for each matching packet
2   -A INPUT -s 10.1.34.40 -j DROP # also drop each matching packet
```

It is a good practice to create a custom chain, which logs and drops any packet passed to it, for this same purpose. It removes redundancy from the main chains (by allowing us to write a single rule where earlier there were two). It also improves efficiency because the matching process for the packet occurs only once, instead of twice. Listing 22 shows how to do the same thing as listing 21 using a custom chain.

Listing 22: iptables rules to log and drop packets using a user-defined chain

```
# Create the LOGDROP chain
iptables -N LOGDROP                # create a chain called LOGDROP
iptables -A LOGDROP \\             # append this rule to chain LOGDROP
         -j LOG \\                 # jump to action LOG
         --log-prefix "LOGDROP " # with log-prefix LOGDROP (optional)
iptables -A LOGDROP \\             # append to chain LOGDROP
         -j DROP                   # drop the packet

# Log and drop all packets from 10.1.34.40
iptables -A INPUT -s 10.1.34.40 -j LOGDROP # notice the action specified as LOGDROP
```

The custom-chains method is especially helpful in case of rules where the matching criteria are many, and complex.

3.5.7 IPTables matches

-p, --protocol :

`iptables -A input -p tcp ...`

This match is used to check for certain protocols. Examples of protocols are TCP, UDP and ICMP. The protocol must either be one of the internally specified TCP, UDP or ICMP. It may also take a value specified in the /etc/protocols file, and if it can't find the protocol there it will reply with an error. The protocl may also be an integer value. For example, the ICMP protocol is integer value 1, TCP is 6 and UDP is 17. Finally, it may also take the value ALL. ALL means that it matches only TCP, UDP and ICMP. If this match is given the integer value of zero (0), it means ALL protocols, which in turn is the default behavior, if the `--protocol` match is not used. This match can also be inversed with the ! sign, so `--protocol ! tcp` would mean to match UDP and ICMP.

-s, --src, --source :

`iptables -A INPUT -s 192.168.1.1 ...`

This is the source match, which is used to match packets, based on their source IP address. The main form can be used to match single IP addresses, such as 192.168.1.1. It could also be used with a netmask in a CIDR "bit" form, by specifying the number of ones (1s) on the left side of the network mask. This means that we could for example add /24 to use a 255.255.255.0 netmask. We could then match whole IP ranges, such as our local networks or network segments behind the firewall. The line would then look something like 192.168.0.0/24. This would match all packets in the 192.168.0.x range. Another way is to do it with a regular netmask in the 255.255.255.255 form (i.e., 192.168.0.0/255.255.255.0). We could also invert the match with an ! just as before. If we were, in other words, to use a match in the form of `--source ! 192.168.0.0/24`, we would match all packets with a source address not coming from within the 192.168.0.x range. The default is to match all IP addresses.

-d, --dst, --destination :

`iptables -A INPUT -d 192.168.1.1 ...`

The --destination match is used for packets based on their destination address or addresses. It works pretty much the same as the `--source` match and has the same syntax, except that the match is based on where the packets are going to. To match an IP range, we can add a netmask either in the exact netmask form, or in the number of ones (1's) counted from the

left side of the netmask bits. Examples are: 192.168.0.0/255.255.255.0 and 192.168.0.0/24. Both of these are equivalent. We could also invert the whole match with an ! sign, just as before. `--destination ! 192.168.0.1` would in other words match all packets except those destined to the 192.168.0.1 IP address.

-i, --in-interface :

```
iptables -A INPUT -i eth0 ...
```

This match is used for the interface the packet came in on. Note that this option is only legal in the INPUT, FORWARD and PREROUTING chains and will return an error message when used anywhere else. The default behaviour of this match, if no particular interface is specified, is to assume a string value of +. The + value is used to match a string of letters and numbers. A single + would, in other words, tell the kernel to match all packets without considering which interface it came in on. The + string can also be appended to the type of interface, so eth+ would be all Ethernet devices. We can also invert the meaning of this option with the help of the ! sign. The line would then have a syntax looking something like -i ! eth0, which would match all incoming interfaces, except eth0.

-o, --out-interface :

```
iptables -A FORWARD -o eth0 ...
```

The `--out-interface` match is used for packets on the interface from which they are leaving. Note that this match is only available in the OUTPUT, FORWARD and POSTROUTING chains, the opposite in fact of the `--in-interface` match. Other than this, it works pretty much the same as the `--in-interface` match. The + extension is understood as matching all devices of similar type, so eth+ would match all eth devices and so on. To invert the meaning of the match, you can use the ! sign in exactly the same way as for the `--in-interface` match. If no `--out-interface` is specified, the default behaviour for this match is to match all devices, regardless of where the packet is going.

-f, --fragment :

```
iptables -A INPUT -f ...
```

This match is used to match the second and third part of a fragmented packet. The reason for this is that in the case of fragmented packets, there is no way to tell the source or destination ports of the fragments, nor ICMP types, among other things. Also, fragmented packets might in rather special cases be used to compound attacks against other computers. Packet fragments like this will not be matched by other rules, and hence this match was created. This option can also be used in conjunction with the ! sign; however, in this case the ! sign must precede the match, i.e. ! -f. When this match is inverted, we match all header fragments and/or unfragmented packets. What this means, is that we match all the first fragments of fragmented packets, and not the second, third, and so on. We also match all packets that have not been fragmented during transfer. Note also that there are really good defragmentation options within the kernel that you can use instead. As a secondary note, if you use connection tracking you will not see any fragmented packets, since they are dealt with before hitting any chain or table in iptables.

3.5.8 Implicit matches

Here we describe those matches which are loaded implicitly. Implicit matches are implied, taken for granted, automatic. For example when we match on `--protocol tcp` without any further criteria. There are currently three types of implicit matches for three different protocols. These are TCP matches, UDP matches and ICMP matches. The TCP based matches contain a set of unique criteria that are available only for TCP packets. UDP based matches contain another set of criteria that are available only for UDP packets.

TCP Matches

These matches are protocol specific and are only available when working with TCP packets and streams. To use these matches, one has to specify `--protocol tcp` on the command line before trying to use them. Note that the `--protocol tcp` match must be to the left of the protocol specific matches.

--sport, --source-port :

```
iptables -A INPUT -p tcp --sport 22 ...
```

The `--source-port` match is used to match packets based on their source port. Without it, we imply all source ports. This match can either take a service name or a port number. If we specify a service name, the service name must be in the `/etc/services` file, since iptables uses this file in which to find. If we specify the port by its number, the rule will load slightly faster, since iptables doesn't have to check up the service name. However, the match might be a little bit harder to read than if we use the service name. While writing a rule-set consisting of 200 rules or more (large rule-set), we should definitely use port numbers, since the difference is really noticeable. (On a slow server, this could make as much as 10 seconds difference, for a large rule-set containing 1000 rules or so). We can also use the `--source-port` match to match a range of ports, `--source-port 22:80` for example. This example would match all source ports between 22 and 80. If we omit specifying the first port, port 0 is assumed (is implicit). `--source-port :80` would then match port 0 through 80. And if the last port specification is omitted, port 65535 is assumed. If one were to write `--source-port 22:`, it would signify a match for all ports from port 22 through port 65535. If we invert the port range, iptables automatically reverses the inversion, i.e. `--source-port 80:22` is simply interpreted as `--source-port 22:80`. We can invert a match by adding a ! sign. For example, `--source-port ! 22` means match all ports but port 22. The inversion could also be used together with a port range and would then look like `--source-port ! 22:80`, which would mean we want to match all ports but ports 22 through 80. Note that this match does not handle multiple separated ports and port ranges. For more information about those, look at the multiport match extension.

--dport, --destination-port :

```
iptables -A INPUT -p tcp --dport 22 ...
```

This match is used to match TCP packets according to their destination port. It uses exactly the same syntax as the `--source-port` match. It understands port and port range specifications, as well as inversions. It also reverses high and low ports in port range specifications, as above. The match will also assume values of 0 and 65535 if the high or low port is left out in a port range specification. In other words, exactly the same as the `--source-port` syntax.

--tcp-flags :

```
iptables -p tcp --tcp-flags SYN,FIN,ACK SYN ...
```

This match is used to match on the TCP flags in a packet. First of all, the match takes a list of flags to compare (a mask) and secondly it takes list of flags that should be set to 1, or turned on. Both lists should be comma-delimited. The match knows about the SYN, ACK, FIN, RST, URG, PSH flags, and it also recognizes the words ALL and NONE. ALL and NONE is pretty much self describing: ALL means to use all flags and NONE means to use no flags for the option. `--tcp-flags` ALL NONE would in other words mean to check all of the TCP flags and match if none of the flags are set. This option can also be inverted with the ! sign. For example, if we specify ! SYN,FIN,ACK SYN, we would get a match that would match packets that had the ACK and FIN bits set, but not the SYN bit. Also note that the comma delimitation should not include spaces.

--tcp-option :

```
iptables -p tcp --tcp-option 16 ...
```

This match is used to match packets depending on their TCP options. A TCP Option is a specific part of the header. This part consists of 3 different fields. The first one is 8 bits long and tells us which Options are used in this stream, the second one is also 8 bits long and

tells us how long the options field is. The reason for this length field is that TCP options are, well, optional. To be compliant with the standards, we do not need to implement all options, but instead we can just look at what kind of option it is, and if we do not support it, we just look at the length field and can then jump over this data. This match is used to match different TCP options depending on their decimal values. It may also be inverted with the ! flag, so that the match matches all TCP options but the option given to the match. For a complete list of all options, take a closer look at the Internet Engineering Task Force who maintains a list of all the standard numbers used on the Internet.

UDP Matches

These matches are specific to UDP Packets. They are implicitly loaded when you specify the `--protocol UDP` match and will be available after this specification. Note that UDP packets are not connection oriented, and hence there is no such thing as different flags to set in the packet to give data on what the datagram is supposed to do, such as open or closing a connection, or if they are just simply supposed to send data. UDP packets do not require any kind of acknowledgment either. If they are lost, they are simply lost (Not taking ICMP error messaging etc into account). This means that there are quite a lot less matches to work with on a UDP packet than there is on TCP packets.

--sport, --source-port :

`iptables -A INPUT -p udp --sport 53 ...`

This match works exactly the same as its TCP counterpart.

--dport, --destination-port :

`iptables -A INPUT -p udp --dport 53 ...`

The same goes for this match as for `--source-port` above. It is exactly the same as for the equivalent TCP match, but here it applies to UDP packets.

ICMP Matches

ICMP packets are even more short-lived than UDP Packets, in that they are connectionless. The ICMP protocol is mainly used for error reporting and for connection controlling and such. ICMP is not a protocol subordinated to the IP protocol, but more of a protocol that augments the IP protocol and helps in handling errors. The headers of ICMP packets are very similar to those of the IP headers, but differ in a number of ways. The main feature of this protocol is the type header, that tells us what the packet is for. One example is, if we try to access an unaccessible IP address, we would normally get an ICMP host unreachable in return.

There is only one ICMP specific match available for ICMP packets, and hopefully this should suffice. This match is implicitly loaded when we use the --protocol ICMP match and we get access to it automatically. Note that all the generic matches can also be used, so that among other things we can match on the source and destination addresses.

--icmp-type :

`iptables -A INPUT -p icmp --icmp-type 8 ...`

This match is used to specify the ICMP type to match. ICMP types can be specified either by their numeric values or by their names. Numerical values are specified in RFC 792. To find a complete listing of the ICMP name values, run `iptables --protocol icmp --help`. This match can also be inverted with the ! sign, for example: `--icmp-type ! 8`. Note that some ICMP types are obsolete, and others again may be "dangerous" for an unprotected host since they may, among other things, redirect packets to the wrong places. The type and code may also be specified by their typename, numeric type, and type/code as well. For example `--icmp-type network-redirect`, `--icmp-type 8` or `--icmp-type 8/0`. For a complete listing of the names, run `iptables -p icmp --help` (short version of the command mentioned previously).

3.5.9 Explicit matches

Explicit matches are those that have to be specifically loaded with the `-m` or `--match` option. State matches, for example, demand the directive `-m` state prior to entering the actual match that we want to use. Some of these matches may be protocol specific . Some may be unconnected with any specific protocol, for example connection states. The difference between implicitly loaded matches and explicitly loaded ones, is that the implicitly loaded matches will automatically be loaded when, for example, we match on the properties of TCP packets, while explicitly loaded matches will never be loaded automatically.

Conmark match

The connmark match is used to match marks that has been set on a connection with the CONNMARK target. It only takes one option.

--mark :

`iptables -A INPUT -m connmark --mark 12 -j ACCEPT`

The mark option is used to match a specific mark associated with a connection. The mark match must be exact, and if we want to filter out unwanted flags from the connection mark before actually matching anything, we can specify a mask that will be ANDed to the connection mark. For example, if we have a connection mark set to 33 (10001 in binary) on a connection, and want to match the first bit only, we would be able to run something like `--mark 1/1`. The mask (00001) would be masked to 10001, so 10001 && 00001 equals 1, and then matched against the 1.

Conntrack match

The conntrack match is an extended version of the state match, which makes it possible to match packets in a much more granular way. It let's you look at information directly available in the connection tracking system. There are a number of different matches put together in the conntrack match, for several different fields in the connection tracking system. These are compiled together into the list below. To load these matches, you need to specify `-m conntrack`.

--ctstate :

`iptables -A INPUT -p tcp -m conntrack --ctstate RELATED ...`

This match is used to match the state of a packet, according to the conntrack state. It is used to match pretty much the same states as in the original state match. The valid entries for this match are:

- INVALID
- ESTABLISHED
- NEW
- RELATED
- SNAT
- DNAT

The entries can be used together with each other separated by a comma. For example, `-m conntrack --ctstate ESTABLISHED,RELATED`. It can also be inverted by putting a ! in front of `--ctstate`. For example: `-m conntrack ! --ctstate ESTABLISHED,RELATED`, which matches all but the ESTABLISHED and RELATED states.

--ctproto :

`iptables -A INPUT -p tcp -m conntrack --ctproto TCP ...`

This matches the protocol, the same as the `--protocol` does. It can take the same types of values, and is inverted using the ! sign. For example, `-m conntrack ! --ctproto TCP` matches all protocols but the TCP protocol.

--ctorigsrc :

```
iptables -A INPUT -p tcp -m conntrack --ctorigsrc 192.168.0.0/24 ...
```

`--ctorigsrc` matches based on the original source IP specification of the conntrack entry that the packet is related to. The match can be inverted by using a ! between the `--ctorigsrc` and IP specification, such as `--ctorigsrc ! 192.168.0.1`. It can also take a netmask of the CIDR form, such as `--ctorigsrc 192.168.0.0/24`.

--ctorigdst :

```
iptables -A INPUT -p tcp -m conntrack --ctorigdst 192.168.0.0/24 ...
```

This match is used exactly as the --ctorigsrc, except that it matches on the destination field of the conntrack entry. It has the same syntax in all other respects.

--ctreplsrc :

```
iptables -A INPUT -p tcp -m conntrack --ctreplsrc 192.168.0.0/24 ...
```

The `--ctreplsrc` match is used to match based on the original conntrack reply source of the packet. Basically, this is the same as the `--ctorigsrc`, but instead we match the reply source expected of the upcoming packets. This target can, of course, be inverted and address a whole range of addresses, just the same as the the previous targets in this class.

--ctrepldst :

```
iptables -A INPUT -p tcp -m conntrack --ctrepldst 192.168.0.0/24 ...
```

The `--ctrepldst` match is the same as the `--ctreplsrc` match, with the exception that it matches the reply destination of the conntrack entry that matched the packet. It too can be inverted, and accept ranges, just as the `--ctreplsrc` match.

--ctstatus :

```
iptables -A INPUT -p tcp -m conntrack --ctstatus RELATED ...
```

This matches the status of the connection. It can match the following statuses:

- NONE - The connection has no status at all.
- EXPECTED - This connection is expected and was added by one of the expectation handlers.
- SEEN_REPLY - This connection has seen a reply but isn't assured yet.
- ASSURED - The connection is assured and will not be removed until it times out or the connection is closed by either end.

This can also be inverted by using the ! sign.

Limit match

The limit match extension must be loaded explicitly with the `-m limit` option. This match can, for example, be used to advantage to give limited logging of specific rules etc. For example, we could use this to match all packets that do not exceed a given value, and after this value has been exceeded, limit logging of the event in question. Think of a time limit: We could limit how many times a certain rule may be matched in a certain time frame, for example to lessen the effects of DoS syn flood attacks. This is its main usage, but there are more usages, of course. The limit match may also be inverted by adding a ! flag in front of the limit match. It would then be expressed as `-m limit ! --limit 5/s`.This means that all packets will be matched after they have broken the limit.

To further explain the limit match, it is basically a token bucket filter (like delay pools in squid). Consider having a leaky bucket where the bucket leaks X packets per time-unit. X is defined depending on how many matching packets we get, so if we get 3 packets, the bucket leaks 3 packets per that time-unit. The `--limit` option tells us how many packets to refill the bucket with per time-unit, while the `--limit-burst` option tells us how big the bucket is in the first place. So, setting `--limit 3/minute --limit-burst 5`, and then receiving 5 matches will

empty the bucket. After 20 seconds, the bucket is refilled with another token, and so on until the `--limit-burst` is reached again or until they get used.

Let us consider the example below for further explanation of how this may look.

1. We set a rule with `-m limit --limit 5/second --limit-burst 10/second`. The limit-burst token bucket is set to 10 initially. Each packet that matches the rule uses a token.
2. We get a packet that matches, 1-2-3-4-5-6-7-8-9-10, all within a 1/1000 of a second.
3. The token bucket is now empty. Once the token bucket is empty, the packets that qualify for the rule otherwise no longer match the rule and proceed to the next rule if any, or hit the chain policy.
4. For each 1/5 s without a matching packet, the token count goes up by 1, upto a maximum of 10. 1 second after receiving the 10 packets, we will once again have 5 tokens left.
5. And of course, the bucket will be emptied by 1 token for each packet it receives.

Limit match options:

--limit :

```
iptables -A INPUT -m limit --limit 3/hour ...
```

This sets the maximum average match rate for the limit match. It is specified with a number and an optional time unit. The following time units are currently recognized: second/minute/hour/day. The default value here is 3 per hour, or `3/hour`. This tells the limit match how many times to allow the match to occur per time unit (e.g. per minute).

--limit-burst :

```
iptables -A INPUT -m limit --limit-burst 5 ...
```

This is the setting for the burst limit of the limit match. It tells iptables the maximum number of tokens available in the bucket when we start, or when the bucket is full. This number gets decremented by one for every packet that arrives, down to the lowest possible value, 1. The bucket will be refilled by the limit value every time unit, as specified by the `--limit` option. The default `--limit-burst value` is 5.

Mark match

The mark match extension is used to match packets based on the marks they have set. A mark is a special field, only maintained within the kernel, that is associated with the packets as they travel through the computer. Marks may be used by different kernel routines for such tasks as traffic shaping and filtering.

Mark match options:

--mark :

```
iptables -t mangle -A INPUT -m mark --mark 1 ...
```

This match is used to match packets that have previously been marked. Marks can be set with the MARK target. All packets traveling through Netfilter get a special mark field associated with them. Note that this mark field is not in any way propagated, within or outside the packet. It stays inside the computer that made it. If the mark field matches the mark, it is a match. The mark field is an unsigned integer, hence there can be a maximum of 4294967296 different marks. We may also use a mask with the mark. The mark specification would then look like, for example, `--mark 1/1`. If a mask is specified, it is logically ANDed with the mark specified before the actual comparison.

Recent match

The recent match is a rather large and complex matching system, which allows us to match packets based on recent events that we have previously matched. For example, if we would see an outgoing

IRC connection, we could set the IP addresses into a list of hosts, and have another rule that allows identd requests back from the IRC server within 15 seconds of seeing the original packet.

Before we can take a closer look at the match options, let's try and explain a little bit how it works. First of all, we use several different rules to accomplish the use of the recent match. The recent match uses several different lists of recent events. The default list being used is the DEFAULT list. We create a new entry in a list with the set option, so once a rule is entirely matched (the set option is always a match), we also add an entry in the recent list specified. The list entry contains a timestamp, and the source IP address used in the packet that triggered the set option. Once this has happened, we can use a series of different recent options to match on this information, as well as update the entries timestamp, etc.

Finally, if we would for some reason want to remove a list entry, we would do this using the --remove match option from the recent match. All rules using the recent match, must load the recent module (-m recent) as usual. Before we go on with an example of the recent match, let's take a look at some of the options.

Recent match options (relevant ones):

--set :

```
iptables -A OUTPUT -m recent --set ...
```

This creates a new list entry in the named recent list, which contains a timestamp and the source IP address of the host that triggered the rule. This match will always return success, unless it is preceded by a ! sign, in which case it will return failure.

--rcheck :

```
iptables -A OUTPUT -m recent --name examplelist --rcheck ...
```

The --rcheck option will check if the source IP address of the packet is in the named list. If it is, the match will return true, otherwise it returns false. The option may be inverted by using the ! sign. In the later case, it will return true if the source IP address is not in the list, and false if it is in the list.

--update :

```
iptables -A OUTPUT -m recent --name examplelist --update ...
```

This match is true if the source combination is available in the specified list and it also updates the last-seen time in the list. This match may also be reversed by setting the ! mark in front of the match. For example, ! --update.

--remove :

```
iptables -A INPUT -m recent --name example --remove ...
```

This match will try to find the source address of the packet in the list, and returns true if the packet is there. It will also remove the corresponding list entry from the list. The command is also possible to inverse with the ! sign.

--seconds :

```
iptables -A INPUT -m recent --name example --check --seconds 60 ...
```

This match is only valid together with the --check and --update matches. The --seconds match is used to specify how long since the "last seen" column was updated in the recent list. If the last seen column was older than this amount in seconds, the match returns false. Other than this the recent match works as normal, so the source address must still be in the list for a true return of the match.

--hitcount :

```
iptables -A INPUT -m recent --name example --check --hitcount 20 ...
```

The --hitcount match must be used together with the --check or --update matches and it will limit the match to only include packets that have seen at least the hitcount amount of packets. If this match is used together with the --seconds match, it will require the specified

hitcount packets to be seen in the specific timeframe. This match may also be reversed by adding a ! sign in front of the match. Together with the --seconds match, this means that a maximum of this amount of packets may have been seen during the specified timeframe. If both of the matches are inversed, then a maximum of this amount of packets may have been seen during the last minumum of seconds.

--rsource :

```
iptables -A INPUT -m recent --name example --rsource ...
```

The --rsource match is used to tell the recent match to save the source address and port in the recent list. This is the default behavior of the recent match.

--rdest :

```
iptables -A INPUT -m recent --name example --rdest ...
```

The --rdest match is the opposite of the --rsource match in that it tells the recent match to save the destination address and port to the recent list.

3.6 Some recommended Best Practices for iptables

Don't set the default policy to DROP

All iptables chains have a default policy setting. If a packet doesn't match any of the rules in a relevant chain, it will match the default policy and will be handled accordingly. Setting the default policy to DROP can bring about some unintended consequences.

Consider a situation where the INPUT chain contains quite a few rules allowing traffic, and the default policy is set to DROP. Later on, another administrator logs into the server and flushes the rules. This will render the server completely inaccessible immediately. All of the packets will be dropped since they match the default policy in the chain.

Instead of using the default policy, it is recommended to place an explicit DROP/REJECT rule at the bottom of the chain that matches everything. Thus the default policy can be left as ACCEPT and this should reduce the chance of blocking all access to the server.

Remember `localhost`

Lots of applications require access to the lo interface. We must ensure that rules are setup so that the `lo` interface is not disturbed.

Split complicated rule groups into separate chains

It's important to keep your iptables rules manageable. If you have a certain subset of rules that may be a little complicated, consider breaking them out into their own chain. You can just add in a jump to that chain from your default set of chains.

Use REJECT until you know your rules are working properly

When you're writing iptables rules, you'll probably be testing them pretty often. One way to speed up that process is to use the REJECT target rather than DROP. You'll get an immediate rejection of your traffic (a TCP reset) instead of wondering if your packet is being dropped or if it's making it to your server at all. Once you're done with your testing, you can flip the rules from REJECT to DROP if you prefer.

Be stringent with your rules

Try to make your rules as specific as possible for your needs. For example, to allow ICMP pings on servers (so that network tests can be run against them), one could easily add a rule into the INPUT chain that looks like the one in listing 23.

Listing 23: Allow ping – non-stringent

```
1  iptables -A INPUT -p icmp -m icmp -j ACCEPT
```

But it isn't prudent to simply allow all ICMP traffic. There are many types of ICMP Control Messages, but for our purpose here, only allowing echo-requests is sufficient (listing 24).

Listing 24: Allow ping – stringent

```
1 iptables -A INPUT -p icmp -m icmp --icmp-type 8 -j ACCEPT
```

Use comments for obscure rules

We often write rules to cover edge cases that other administrators might not understand. It is always wise to add comments explaining such rules.

Comments can be added using the `-m comment` directive, as shown in listing 25. However, it only accepts comments upto 256 chars in size.

Listing 25: Adding comments – the right way

```
1 iptables -A INPUT -s www.spamhost.org -m comment --comment "block spamhost.org" -j DROP
```

These comments will appear in the iptables output on listing the current rules. They will also appear in the saved iptables rules.

4 Applications

The applications of iptables are numerous. The following are some of the primary packet processing techniques and their applications. Some of these have dedicated modules for their purpose, others can be accomplished using the basic iptables functionalities :

Packet Filtering
: Packet filtering is the most basic type of network packet processing. Packet filtering involves examining packets at various points as they move through the kernel's networking code and making decisions about how the packets should be handled (accepted into the next stage of processing, dropped completely without a reply, rejected with a reply, and so on).

Accounting
: Accounting, as the name suggests, involves using byte and/or packet counters associated with packet matching criteria to monitor network traffic volumes.

Connection Tracking
: Connection tracking provides additional information that can match related packets in ways that are otherwise impossible. For example, FTP (file transfer protocol) sessions can involve two separate connections: one for control and one for data transfer. Connection tracking for FTP monitors the control connection and uses knowledge of the FTP protocol to extract enough information from the control interactions to identify the data connections when they are created. This tracking information is then made available for use by packet processing rules.

Packet mangling
: Packet mangling involves making changes to packet header fields (such as network addresses and port numbers) or payloads.

Network Address Translation (NAT)
: Network address translation is a type of packet mangling that involves overwriting the source and/or destination addresses and/or port numbers. Connection tracking information is used to mangle related packets in specific ways. The term "Source NAT" (or just S-NAT or SNAT) refers to NAT involving changes to the source address and/or port, and "Destination NAT" (or just D-NAT or DNAT) refers to NAT involving changes to the destination address and/or port.

Masquerading
: Masquerading is a special type of SNAT in which one computer rewrites packets to make them appear to come from itself. The computer's IP address used is determined automatically, and if it changes, old connections are destroyed appropriately. Masquerading is commonly used to share an Internet connection with a dynamic IP address among a network of computers.

Port Forwarding
: Port forwarding is a type of DNAT in which one com- puter (such as a firewall) acts as a proxy for one or more other computers. The firewall accepts packets addressed to itself from the outside network, but rewrites them to appear to be addressed to other computers on the inside network before sending them on to their new destinations. In addition, related reply packets from the inside computers are rewritten to appear to be from the firewall and sent back to the appropriate outside computer. Port forwarding is commonly used to provide publicly accessible network services (such as web or email servers) by computers other than the firewall, without requiring more than one public IP address. To the outside world, it appears that the services are being provided by the proxy machine, and to the actual server, it appears that all requests are coming from the proxy machine.

Load Balancing
: Load balancing involves distributing connections across a group of servers so that higher total throughput can be achieved. One way to implement simple load balancing is to set up port forwarding so that the destination address is selected in a round-robin fashion from a list of possible destinations.

5 Some specific techniques and their configuration

5.1 Port Knocking

Port knocking is a technique to secure ssh (or any other methods) access to the machine. It involves knowing a pre-determined set of ports, which when "knocked" in sequence, open the ssh port for the knocking IP. In other words, without performing a valid knock (telnet based connection attempt on the set of ports, in the correct order), one cannot "unlock" the required port.

Portknocking is a a stealthy and robust system for network authentication across closed ports. For instance, this can be used to avoid brute force attacks to ssh or ftp services.

Port knocking can be implemented using iptables in a number of ways, two of which we will discuss here.

5.1.1 Implementation using custom-chains

this method uses the recent iptables module, to keep track of the ports knocked, and act accordingly.

We will demonstrate how to setup a 4-port knocking system wherein the required ports to be knocked are 10000,10001,10002 and 10003. Also, the user gets 30 seconds over which to perform the knocking, otherwise he must start from stage 1.

Listing 26: iptables rules to setup port knocking

```
-N STAGE2
-A STAGE2 -m recent --name STAGE1 --remove
-A STAGE2 -m recent --name STAGE2 --set
-A STAGE2 -j LOG --log-prefix "INTO STAGE2: "

-N STAGE3
-A STAGE3 -m recent --name STAGE2 --remove
```

```
-A STAGE3 -m recent --name STAGE3 --set
-A STAGE3 -j LOG --log-prefix "INTO STAGE3: "

-N STAGE4
-A STAGE4 -m recent --name STAGE3 --remove
-A STAGE4 -m recent --name STAGE4 --set
-A STAGE4 -j LOG --log-prefix "INTO STAGE4: "

-A INPUT -m recent --update --name STAGE1

-A INPUT -p tcp --dport 10000 -m recent --set --name STAGE1
-A INPUT -p tcp --dport 10001 -m recent --rcheck --name STAGE1 -j STAGE2
-A INPUT -p tcp --dport 10002 -m recent --rcheck --name STAGE2 -j STAGE3
-A INPUT -p tcp --dport 10003 -m recent --rcheck --name STAGE3 -j STAGE4

-A INPUT -p tcp --dport 22 -m recent --rcheck --seconds 30 --name STAGE4 -j ACCEPT
```

5.1.2 Using the Portknock0 project iptables module

The PortKnock0 Project is composed of two parts: an iptables extension (user space) and a netfilter extension (kernel space). Both modules are used to implement Port Knocking

Following the steps described at `http://portknocko.berlios.de/README.html`, one can install PortKnock0. Once installed, it provides a new iptables module, using which portknocking can be setup very easily.

Listing 27 demonstrates setting up the same config as shown earlier, with custom chains.

Listing 27: iptables rules to setup port knocking with PortKnock0

```
iptables -A INPUT \\                           # append to chain INPUT
         -p tcp \\                             # tcp packets
         -m state \\                           # load module state
         --state NEW \\                        # only match NEW connections
         -m pknock \\                          # load module pknock
         --knockports 10000,10001,10002,10003 \\# the ports for knocking
         --name SSH \\                         # name
         --time 30 \\                          # max. allowed time between knocks
         --strict \\                           # if the user fails one knock in the
             sequence he/she must start over
         -m tcp \\                             # load module tcp
         --dport 22 \\                         # destination port 22 (SSH)
         -j ACCEPT                             # ACCEPT if all matches are OK
```

The machine can then be accessed using telnet to knock the ports and then regular SSH (listing 28).

Listing 28: Accessing the machine

```
$ ssh user@someserver     # won't work

$ telnet someserver 10000 # first knock
$ telnet someserver 10001
$ telnet someserver 10002
$ telnet someserver 10003 # last knock

$ ssh user@someserver     # will work now
```

5.2 Rate Limiting

iptables is commonly used for rate limiting, for bandwidth considerations. Rate limiting is also used in various ways to secure the system, with iptables. Some of the relevant applications are as described here.

5.2.1 Protecting against ping flood attacks

Here the limit module is used to keep check on the icmp echo-requests that are characteristic of ping. The following rules setup reasonably tight ping flood protection :

Listing 29: iptables rules to protect against a ping flood attack

```
1 -A INPUT -p icmp --icmp-type echo-request -m limit --limit 60/minute --limit-burst
2 120 -j ACCEPT
3 -A INPUT -p icmp --icmp-type echo-request -m limit --limit 1/minute --limit-burst 2
4 -j LOG
5 -A INPUT -p icmp --icmp-type echo-request -j DROP
```

5.3 Connection Limiting

Connection limiting also has various uses. The connlimit module is used for connection limiting.

5.3.1 Limiting the number of SSH connections from a host (in parallel)

Often it is required to restrict the number of parallel ssh connections a user can make, to the server. This can be effectively accomplished using iptables.

Listing 30: iptables rules to limit parallel SSH connections to 4 with logging

```
1 -A INPUT -p tcp --syn --dport 22 -m connlimit --connlimit-above 4 -m limit --limit
2 1/minute --limit-burst 2 -j LOG
3 -A INPUT -p tcp --syn --dport 22 -m connlimit --connlimit-above 4 -j REJECT
```

5.3.2 Bruteforce attack protection

This can be done by not allowing more than a specified number of connections, over a specific time interval, from any IP.

For example, in listing we have configured things so as to not allow more then 10 connection attempts within 60 seconds from any IP.

Listing 31: iptables rules to prevent more than 10 connections within 60 seconds from any IP

```
1 -A INPUT -m state --state NEW -m tcp -p tcp --dport 22 -m recent --name ssh_limit
2 --set
3 -A INPUT -m state --state NEW -m tcp -p tcp --dport 22 -m recent --name ssh_limit
4 --rcheck --seconds 60 --hitcount 10 -m limit --limit 1/minute --limit-burst 2 -j LOG
5 -A INPUT -m state --state NEW -m tcp -p tcp --dport 22 -m recent --name ssh_limit
6 --rcheck --seconds 60 --hitcount 10 -j REJECT --reject-with icmp-host-prohibited
```

6 iptables at IIIT-H

IIIT-H Servers use iptables as their firewall mechanism. Most servers that interface with the outside world have iptables configured quite thoroughly. However, after some analysis of the configuration, certain improvements and optimizations can be suggested.

6.1 Suggestions for Optimization and Improvement

6.1.1 Optimizing iptables by creating user-defined chains

iptables allows for very complex and lengthy rulesets. Improper rule structure can lead to inefficiency in your packet filtering system which can in turn decrease effective bandwidth and serving capabilities. It is important that you carefully consider the order and structure of your packet filter layout.

The ultimate goal of packet filtering is to control and limit traffic to only that which you desire to accept, send, and forward. The secondary goal is to get each packet out of iptables as soon as possible by placing the packet on an ACCEPT or DROP target. While secondary, inefficiencies in your iptables structure can render the packet filtering capabilities useless as effectively throttling your bandwidth.

iptables allows you to create your own chains and add them as targets from rules on other (including the default set of) chains. This allows us to create a shallow, wide decision tree instead of a deep and narrow. While keeping in mind that if a packet gets incorrectly filtered, the whole system is useless, the shallower a decision tree, the faster the packets will be filtered.

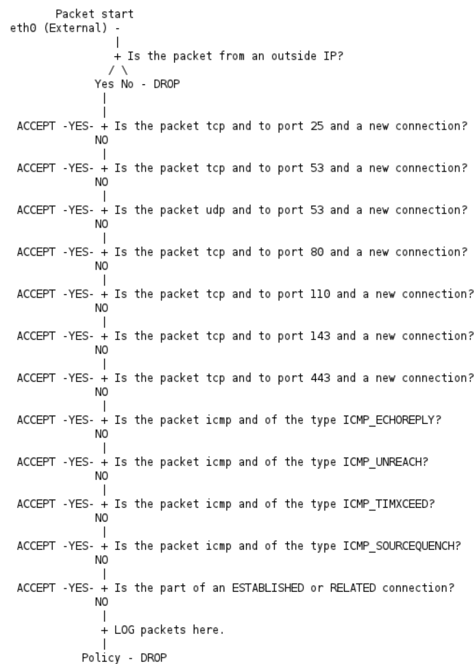

Figure 6: Deep, Narrow Chain decision tree

The chains whose decision trees are shown in figures 6 and 7 accomplish the exact same thing. However, the second one is more efficient.

6.1.2 Reordering rules based on counters

In a chain, we must always try to place the *hottest* rules at the top. What this means is, the rules which have the maximum likelihood of being matched, overall, should come first, and so on.

This ensures that each time a new packet is analyzed, the chances of it reaching the lower ends of the chain are less. This improves efficiency.

One standard method of finding out whether to re-think your sequence of rules is to check the hit counters that iptables maintains, using `iptables -L -n -v` and push the rules with higher

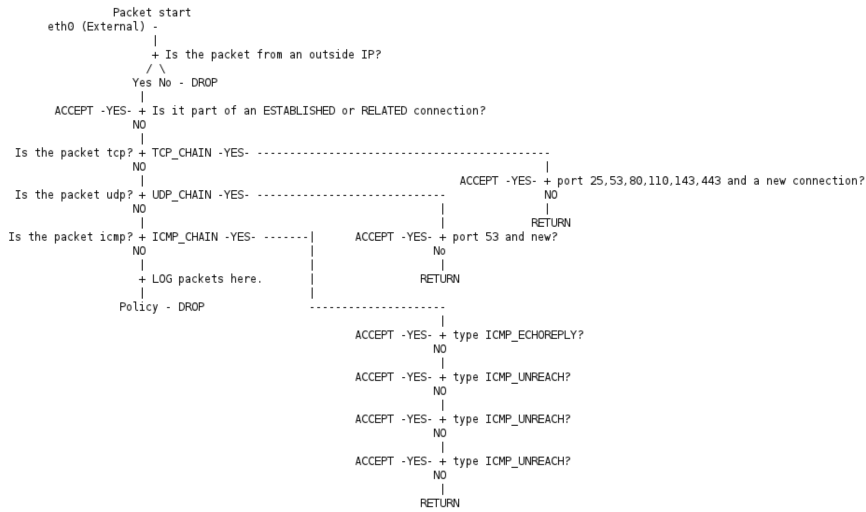

Figure 7: Shallower, Wider Chain decision tree

counts towards the top.

6.1.3 Logging and Dropping/Accepting with a single rule

As described earlier, in Subsubsection 3.5.6, we can add custom chains to reduce the amount of processing for packets that have to be logged *and* accepted/dropped.

7 Man Pages

The rest of this document contains the man-pages which are relevant to the text. Kindly consult them for reference.

The included man-pages are:

- `iptables`
- `iptables-save`
- `iptables-restore`

NAME

iptables — administration tool for IPv4 packet filtering and NAT

SYNOPSIS

iptables [**–t** *table*] {**–A**|**–D**} *chain rule-specification*

iptables [**–t** *table*] **–I** *chain* [*rulenum*] *rule-specification*

iptables [**–t** *table*] **–R** *chain rulenum rule-specification*

iptables [**–t** *table*] **–D** *chain rulenum*

iptables [**–t** *table*] **–S** [*chain* [*rulenum*]]

iptables [**–t** *table*] {**–F**|**–L**|**–Z**} [*chain* [*rulenum*]] [*options...*]

iptables [**–t** *table*] **–N** *chain*

iptables [**–t** *table*] **–X** [*chain*]

iptables [**–t** *table*] **–P** *chain target*

iptables [**–t** *table*] **–E** *old-chain-name new-chain-name*

rule-specification = [*matches...*] [*target*]

match = **–m** *matchname* [*per-match-options*]

target = **–j** *targetname* [*per–target–options*]

DESCRIPTION

Iptables is used to set up, maintain, and inspect the tables of IPv4 packet filter rules in the Linux kernel. Several different tables may be defined. Each table contains a number of built-in chains and may also contain user-defined chains.

Each chain is a list of rules which can match a set of packets. Each rule specifies what to do with a packet that matches. This is called a 'target', which may be a jump to a user-defined chain in the same table.

TARGETS

A firewall rule specifies criteria for a packet and a target. If the packet does not match, the next rule in the chain is the examined; if it does match, then the next rule is specified by the value of the target, which can be the name of a user-defined chain or one of the special values **ACCEPT**, **DROP**, **QUEUE** or **RETURN**.

ACCEPT means to let the packet through. **DROP** means to drop the packet on the floor. **QUEUE** means to pass the packet to userspace. (How the packet can be received by a userspace process differs by the particular queue handler. 2.4.x and 2.6.x kernels up to 2.6.13 include the **ip_queue** queue handler. Kernels 2.6.14 and later additionally include the **nfnetlink_queue** queue handler. Packets with a target of QUEUE will be sent to queue number '0' in this case. Please also see the **NFQUEUE** target as described later in this man page.) **RETURN** means stop traversing this chain and resume at the next rule in the previous (calling) chain. If the end of a built-in chain is reached or a rule in a built-in chain with target **RETURN** is matched, the target specified by the chain policy determines the fate of the packet.

TABLES

There are currently three independent tables (which tables are present at any time depends on the kernel configuration options and which modules are present).

–t, **––table** *table*

This option specifies the packet matching table which the command should operate on. If the kernel is configured with automatic module loading, an attempt will be made to load the appropriate module for that table if it is not already there.

The tables are as follows:

filter: This is the default table (if no –t option is passed). It contains the built-in chains **INPUT** (for packets destined to local sockets), **FORWARD** (for packets being routed through the box), and **OUTPUT** (for locally-generated packets).

nat: This table is consulted when a packet that creates a new connection is encountered. It consists of three built-ins: **PREROUTING** (for altering packets as soon as they come in), **OUTPUT** (for altering locally-generated packets before routing), and **POSTROUTING** (for altering packets as they are about to go out).

mangle:
This table is used for specialized packet alteration. Until kernel 2.4.17 it had two built-in chains: **PREROUTING** (for altering incoming packets before routing) and **OUTPUT** (for altering locally-generated packets before routing). Since kernel 2.4.18, three other built-in chains are also supported: **INPUT** (for packets coming into the box itself), **FORWARD** (for altering packets being routed through the box), and **POSTROUTING** (for altering packets as they are about to go out).

raw: This table is used mainly for configuring exemptions from connection tracking in combination with the NOTRACK target. It registers at the netfilter hooks with higher priority and is thus called before ip_conntrack, or any other IP tables. It provides the following built-in chains: **PREROUTING** (for packets arriving via any network interface) **OUTPUT** (for packets generated by local processes)

OPTIONS

The options that are recognized by **iptables** can be divided into several different groups.

COMMANDS

These options specify the desired action to perform. Only one of them can be specified on the command line unless otherwise stated below. For long versions of the command and option names, you need to use only enough letters to ensure that **iptables** can differentiate it from all other options.

–A, **––append** *chain rule-specification*
Append one or more rules to the end of the selected chain. When the source and/or destination names resolve to more than one address, a rule will be added for each possible address combination.

–D, **––delete** *chain rule-specification*
–D, **––delete** *chain rulenum*
Delete one or more rules from the selected chain. There are two versions of this command: the rule can be specified as a number in the chain (starting at 1 for the first rule) or a rule to match.

–I, **––insert** *chain* [*rulenum*] *rule-specification*
Insert one or more rules in the selected chain as the given rule number. So, if the rule number is 1, the rule or rules are inserted at the head of the chain. This is also the default if no rule number is specified.

–R, **––replace** *chain rulenum rule-specification*
Replace a rule in the selected chain. If the source and/or destination names resolve to multiple addresses, the command will fail. Rules are numbered starting at 1.

–L, **––list** [*chain*]
List all rules in the selected chain. If no chain is selected, all chains are listed. Like every other iptables command, it applies to the specified table (filter is the default), so NAT rules get listed by
iptables –t nat –n –L
Please note that it is often used with the **–n** option, in order to avoid long reverse DNS lookups. It is legal to specify the **–Z** (zero) option as well, in which case the chain(s) will be atomically listed and zeroed. The exact output is affected by the other arguments given. The exact rules are suppressed until you use
iptables –L –v

–S, **––list–rules** [*chain*]
Print all rules in the selected chain. If no chain is selected, all chains are printed like iptables-save. Like every other iptables command, it applies to the specified table (filter is the default).

−F, **−−flush** [*chain*]
Flush the selected chain (all the chains in the table if none is given). This is equivalent to deleting all the rules one by one.

−Z, **−−zero** [*chain* [*rulenum*]]
Zero the packet and byte counters in all chains, or only the given chain, or only the given rule in a chain. It is legal to specify the **−L**, **−−list** (list) option as well, to see the counters immediately before they are cleared. (See above.)

−N, **−−new−chain** *chain*
Create a new user-defined chain by the given name. There must be no target of that name already.

−X, **−−delete−chain** [*chain*]
Delete the optional user-defined chain specified. There must be no references to the chain. If there are, you must delete or replace the referring rules before the chain can be deleted. The chain must be empty, i.e. not contain any rules. If no argument is given, it will attempt to delete every non-builtin chain in the table.

−P, **−−policy** *chain target*
Set the policy for the chain to the given target. See the section **TARGETS** for the legal targets. Only built-in (non-user-defined) chains can have policies, and neither built-in nor user-defined chains can be policy targets.

−E, **−−rename−chain** *old−chain new−chain*
Rename the user specified chain to the user supplied name. This is cosmetic, and has no effect on the structure of the table.

−h Help. Give a (currently very brief) description of the command syntax.

PARAMETERS

The following parameters make up a rule specification (as used in the add, delete, insert, replace and append commands).

[**!**] **−p**, **−−protocol** *protocol*
The protocol of the rule or of the packet to check. The specified protocol can be one of **tcp**, **udp**, **udplite**, **icmp**, **esp**, **ah**, **sctp** or **all**, or it can be a numeric value, representing one of these protocols or a different one. A protocol name from /etc/protocols is also allowed. A "!" argument before the protocol inverts the test. The number zero is equivalent to **all**. Protocol **all** will match with all protocols and is taken as default when this option is omitted.

[**!**] **−s**, **−−source** *address*[/*mask*][**,**...]
Source specification. *Address* can be either a network name, a hostname, a network IP address (with /*mask*), or a plain IP address. Hostnames will be resolved once only, before the rule is submitted to the kernel. Please note that specifying any name to be resolved with a remote query such as DNS is a really bad idea. The *mask* can be either a network mask or a plain number, specifying the number of 1's at the left side of the network mask. Thus, a mask of *24* is equivalent to *255.255.255.0*. A "!" argument before the address specification inverts the sense of the address. The flag **−−src** is an alias for this option. Multiple addresses can be specified, but this will **expand to multiple rules** (when adding with −A), or will cause multiple rules to be deleted (with −D).

[**!**] **−d**, **−−destination** *address*[/*mask*][**,**...]
Destination specification. See the description of the **−s** (source) flag for a detailed description of the syntax. The flag **−−dst** is an alias for this option.

−j, **−−jump** *target*
This specifies the target of the rule; i.e., what to do if the packet matches it. The target can be a user-defined chain (other than the one this rule is in), one of the special builtin targets which decide the fate of the packet immediately, or an extension (see **EXTENSIONS** below). If this option is omitted in a rule (and **−g** is not used), then matching the rule will have no effect on the packet's fate, but the counters on the rule will be incremented.

−g, **−−goto** *chain*
This specifies that the processing should continue in a user specified chain. Unlike the −−jump option return will not continue processing in this chain but instead in the chain that called us via −−jump.

[**!**] **−i**, **−−in−interface** *name*
Name of an interface via which a packet was received (only for packets entering the **INPUT**, **FORWARD** and **PREROUTING** chains). When the "!" argument is used before the interface name, the sense is inverted. If the interface name ends in a "+", then any interface which begins with this name will match. If this option is omitted, any interface name will match.

[**!**] **−o**, **−−out−interface** *name*
Name of an interface via which a packet is going to be sent (for packets entering the **FORWARD**, **OUTPUT** and **POSTROUTING** chains). When the "!" argument is used before the interface name, the sense is inverted. If the interface name ends in a "+", then any interface which begins with this name will match. If this option is omitted, any interface name will match.

[**!**] **−f**, **−−fragment**
This means that the rule only refers to second and further fragments of fragmented packets. Since there is no way to tell the source or destination ports of such a packet (or ICMP type), such a packet will not match any rules which specify them. When the "!" argument precedes the "−f" flag, the rule will only match head fragments, or unfragmented packets.

−c, **−−set−counters** *packets bytes*
This enables the administrator to initialize the packet and byte counters of a rule (during **INSERT**, **APPEND**, **REPLACE** operations).

OTHER OPTIONS

The following additional options can be specified:

−v, **−−verbose**
Verbose output. This option makes the list command show the interface name, the rule options (if any), and the TOS masks. The packet and byte counters are also listed, with the suffix 'K', 'M' or 'G' for 1000, 1,000,000 and 1,000,000,000 multipliers respectively (but see the **−x** flag to change this). For appending, insertion, deletion and replacement, this causes detailed information on the rule or rules to be printed.

−n, **−−numeric**
Numeric output. IP addresses and port numbers will be printed in numeric format. By default, the program will try to display them as host names, network names, or services (whenever applicable).

−x, **−−exact**
Expand numbers. Display the exact value of the packet and byte counters, instead of only the rounded number in K's (multiples of 1000) M's (multiples of 1000K) or G's (multiples of 1000M). This option is only relevant for the **−L** command.

−−line−numbers
When listing rules, add line numbers to the beginning of each rule, corresponding to that rule's position in the chain.

−−modprobe=*command*
When adding or inserting rules into a chain, use *command* to load any necessary modules (targets, match extensions, etc).

MATCH EXTENSIONS

iptables can use extended packet matching modules. These are loaded in two ways: implicitly, when **−p** or **−−protocol** is specified, or with the **−m** or **−−match** options, followed by the matching module name; after these, various extra command line options become available, depending on the specific module. You can specify multiple extended match modules in one line, and you can use the **−h** or **−−help** options after the module has been specified to receive help specific to that module.

The following are included in the base package, and most of these can be preceded by a "**!**" to invert the

sense of the match.

addrtype

This module matches packets based on their **address type.** Address types are used within the kernel networking stack and categorize addresses into various groups. The exact definition of that group depends on the specific layer three protocol.

The following address types are possible:

UNSPEC
an unspecified address (i.e. 0.0.0.0)

UNICAST
an unicast address

LOCAL
a local address

BROADCAST
a broadcast address

ANYCAST
an anycast packet

MULTICAST
a multicast address

BLACKHOLE
a blackhole address

UNREACHABLE
an unreachable address

PROHIBIT
a prohibited address

THROW
FIXME

NAT FIXME

XRESOLVE

[**!**] **−−src−type** *type*
Matches if the source address is of given type

[**!**] **−−dst−type** *type*
Matches if the destination address is of given type

−−limit−iface−in
The address type checking can be limited to the interface the packet is coming in. This option is only valid in the **PREROUTING**, **INPUT** and **FORWARD** chains. It cannot be specified with the **−−limit−iface−out** option.

−−limit−iface−out
The address type checking can be limited to the interface the packet is going out. This option is only valid in the **POSTROUTING**, **OUTPUT** and **FORWARD** chains. It cannot be specified with the **−−limit−iface−in** option.

ah

This module matches the SPIs in Authentication header of IPsec packets.

[**!**] **−−ahspi** *spi*[**:***spi*]

cluster

Allows you to deploy gateway and back-end load-sharing clusters without the need of load-balancers.

This match requires that all the nodes see the same packets. Thus, the cluster match decides if this node has

to handle a packet given the following options:

−−cluster−total−nodes *num*
Set number of total nodes in cluster.

[**!**] **−−cluster−local−node** *num*
Set the local node number ID.

[**!**] **−−cluster−local−nodemask** *mask*
Set the local node number ID mask. You can use this option instead of **−−cluster−local−node**.

−−cluster−hash−seed *value*
Set seed value of the Jenkins hash.

Example:

iptables −A PREROUTING −t mangle −i eth1 −m cluster −−cluster−total−nodes 2 −−cluster−local−node 1 −−cluster−hash−seed 0xdeadbeef −j MARK −−set−mark 0xffff

iptables −A PREROUTING −t mangle −i eth2 −m cluster −−cluster−total−nodes 2 −−cluster−local−node 1 −−cluster−hash−seed 0xdeadbeef −j MARK -−set−mark 0xffff

iptables −A PREROUTING −t mangle −i eth1 −m mark ! −−mark 0xffff −j DROP

iptables −A PREROUTING −t mangle −i eth2 −m mark ! −−mark 0xffff −j DROP

And the following commands to make all nodes see the same packets:

ip maddr add 01:00:5e:00:01:01 dev eth1

ip maddr add 01:00:5e:00:01:02 dev eth2

arptables −A OUTPUT −o eth1 −−h−length 6 −j mangle −−mangle-mac-s 01:00:5e:00:01:01

arptables −A INPUT −i eth1 −−h-length 6 −−destination-mac 01:00:5e:00:01:01 −j mangle −−mangle−mac−d 00:zz:yy:xx:5a:27

arptables −A OUTPUT −o eth2 −−h−length 6 −j mangle −−mangle−mac−s 01:00:5e:00:01:02

arptables −A INPUT −i eth2 −−h−length 6 −−destination−mac 01:00:5e:00:01:02 −j mangle −−mangle−mac−d 00:zz:yy:xx:5a:27

In the case of TCP connections, pickup facility has to be disabled to avoid marking TCP ACK packets coming in the reply direction as valid.

echo 0 > /proc/sys/net/netfilter/nf_conntrack_tcp_loose

comment

Allows you to add comments (up to 256 characters) to any rule.

−−comment *comment*

Example:
iptables −A INPUT −i eth1 −m comment −−comment "my local LAN"

connbytes

Match by how many bytes or packets a connection (or one of the two flows constituting the connection) has transferred so far, or by average bytes per packet.

The counters are 64-bit and are thus not expected to overflow ;)

The primary use is to detect long-lived downloads and mark them to be scheduled using a lower priority band in traffic control.

The transferred bytes per connection can also be viewed through ‘conntrack −L‘ and accessed via ctnetlink.

NOTE that for connections which have no accounting information, the match will always return false. The "net.netfilter.nf_conntrack_acct" sysctl flag controls whether **new** connections will be byte/packet counted. Existing connection flows will not be gaining/losing a/the accounting structure when be sysctl flag is flipped.

[**!**] **−−connbytes** *from*[**:***to*]
match packets from a connection whose packets/bytes/average packet size is more than FROM and less than TO bytes/packets. if TO is omitted only FROM check is done. "!" is used to match packets not falling in the range.

−−connbytes−dir {**original**|**reply**|**both**}
which packets to consider

−−connbytes−mode {**packets**|**bytes**|**avgpkt**}
whether to check the amount of packets, number of bytes transferred or the average size (in bytes) of all packets received so far. Note that when "both" is used together with "avgpkt", and data is going (mainly) only in one direction (for example HTTP), the average packet size will be about half of the actual data packets.

Example:
iptables .. −m connbytes −−connbytes 10000:100000 −−connbytes−dir both −−connbytes−mode bytes ...

connlimit

Allows you to restrict the number of parallel connections to a server per client IP address (or client address block).

[**!**] **−−connlimit−above** *n*
Match if the number of existing connections is (not) above *n*.

−−connlimit−mask *prefix_length*
Group hosts using the prefix length. For IPv4, this must be a number between (including) 0 and 32. For IPv6, between 0 and 128.

Examples:

allow 2 telnet connections per client host
iptables −A INPUT −p tcp −−syn −−dport 23 −m connlimit −−connlimit−above 2 −j REJECT

you can also match the other way around:
iptables −A INPUT −p tcp −−syn −−dport 23 −m connlimit ! −−connlimit−above 2 −j ACCEPT

limit the number of parallel HTTP requests to 16 per class C sized network (24 bit netmask)
iptables −p tcp −−syn −−dport 80 −m connlimit −−connlimit−above 16 −−connlimit−mask 24 −j REJECT

limit the number of parallel HTTP requests to 16 for the link local network
(ipv6) ip6tables −p tcp −−syn −−dport 80 −s fe80::/64 −m connlimit −−connlimit−above 16 −−connlimit−mask 64 −j REJECT

connmark

This module matches the netfilter mark field associated with a connection (which can be set using the **CONNMARK** target below).

[**!**] **−−mark** *value*[**/***mask*]
Matches packets in connections with the given mark value (if a mask is specified, this is logically ANDed with the mark before the comparison).

conntrack

This module, when combined with connection tracking, allows access to the connection tracking state for this packet/connection.

[**!**] **−−ctstate** *statelist*
statelist is a comma separated list of the connection states to match. Possible states are listed below.

[**!**] **−−ctproto** *l4proto*
Layer-4 protocol to match (by number or name)

[**!**] **−−ctorigsrc** *address*[**/***mask*]

[**!**] **−−ctorigdst** *address*[**/***mask*]

[**!**] **−−ctreplsrc** *address*[**/***mask*]

[**!**] **−−ctrepldst** *address*[**/***mask*]
Match against original/reply source/destination address

[**!**] **−−ctorigsrcport** *port*

[**!**] **−−ctorigdstport** *port*

[**!**] **−−ctreplsrcport** *port*

[**!**] **−−ctrepldstport** *port*
Match against original/reply source/destination port (TCP/UDP/etc.) or GRE key.

[**!**] **−−ctstatus** *statelist*
statuslist is a comma separated list of the connection statuses to match. Possible statuses are listed below.

[**!**] **−−ctexpire** *time*[**:***time*]
Match remaining lifetime in seconds against given value or range of values (inclusive)

−−ctdir {**ORIGINAL**|**REPLY**}
Match packets that are flowing in the specified direction. If this flag is not specified at all, matches packets in both directions.

States for **−−ctstate**:

INVALID
meaning that the packet is associated with no known connection

NEW meaning that the packet has started a new connection, or otherwise associated with a connection which has not seen packets in both directions, and

ESTABLISHED
meaning that the packet is associated with a connection which has seen packets in both directions,

RELATED
meaning that the packet is starting a new connection, but is associated with an existing connection, such as an FTP data transfer, or an ICMP error.

UNTRACKED
meaning that the packet is not tracked at all, which happens if you use the NOTRACK target in raw table.

SNAT A virtual state, matching if the original source address differs from the reply destination.

DNAT A virtual state, matching if the original destination differs from the reply source.

Statuses for **−−ctstatus**:

NONE None of the below.

EXPECTED
This is an expected connection (i.e. a conntrack helper set it up)

SEEN_REPLY
Conntrack has seen packets in both directions.

ASSURED
Conntrack entry should never be early-expired.

CONFIRMED
Connection is confirmed: originating packet has left box.

cpu

[**!**] **−−cpu** *number*

Match cpu handling this packet. cpus are numbered from 0 to NR_CPUS-1 Can be used in combination with RPS (Remote Packet Steering) or multiqueue NICs to spread network traffic on different queues.

Example:

iptables −t nat −A PREROUTING −p tcp −−dport 80 −m cpu −−cpu 0 −j REDIRECT −−to−port 8080

iptables −t nat −A PREROUTING −p tcp −−dport 80 −m cpu −−cpu 1 −j REDIRECT −−to−port 8081

Available since Linux 2.6.36.

dccp

[**!**] **−−source−port**,**−−sport** *port*[**:***port*]

[**!**] **−−destination−port**,**−−dport** *port*[**:***port*]

[**!**] **−−dccp−types** *mask*

Match when the DCCP packet type is one of 'mask'. 'mask' is a comma-separated list of packet types. Packet types are: **REQUEST RESPONSE DATA ACK DATAACK CLOSEREQ CLOSE RESET SYNC SYNCACK INVALID**.

[**!**] **−−dccp−option** *number*

Match if DCP option set.

dscp

This module matches the 6 bit DSCP field within the TOS field in the IP header. DSCP has superseded TOS within the IETF.

[**!**] **−−dscp** *value*

Match against a numeric (decimal or hex) value [0-63].

[**!**] **−−dscp−class** *class*

Match the DiffServ class. This value may be any of the BE, EF, AFxx or CSx classes. It will then be converted into its according numeric value.

ecn

This allows you to match the ECN bits of the IPv4 and TCP header. ECN is the Explicit Congestion Notification mechanism as specified in RFC3168

[**!**] **−−ecn−tcp−cwr**

This matches if the TCP ECN CWR (Congestion Window Received) bit is set.

[**!**] **−−ecn−tcp−ece**

This matches if the TCP ECN ECE (ECN Echo) bit is set.

[**!**] **−−ecn−ip−ect** *num*

This matches a particular IPv4 ECT (ECN-Capable Transport). You have to specify a number between '0' and '3'.

esp

This module matches the SPIs in ESP header of IPsec packets.

[**!**] **−−espspi** *spi*[**:***spi*]

hashlimit

hashlimit uses hash buckets to express a rate limiting match (like the **limit** match) for a group of connections using a **single** iptables rule. Grouping can be done per-hostgroup (source and/or destination address) and/or per-port. It gives you the ability to express "*N* packets per time quantum per group":

matching on source host

"1000 packets per second for every host in 192.168.0.0/16"

matching on source port

"100 packets per second for every service of 192.168.1.1"

matching on subnet

"10000 packets per minute for every /28 subnet in 10.0.0.0/8"

A hash limit option (**−−hashlimit−upto**, **−−hashlimit−above**) and **−−hashlimit−name** are required.

−−hashlimit−upto *amount*[**/second**|**/minute**|**/hour**|**/day**]

Match if the rate is below or equal to *amount*/quantum. It is specified as a number, with an optional time quantum suffix; the default is 3/hour.

−−hashlimit−above *amount*[**/second**|**/minute**|**/hour**|**/day**]

Match if the rate is above *amount*/quantum.

−−hashlimit−burst *amount*

Maximum initial number of packets to match: this number gets recharged by one every time the limit specified above is not reached, up to this number; the default is 5.

−−hashlimit−mode {**srcip**|**srcport**|**dstip**|**dstport**}**,...**

A comma-separated list of objects to take into consideration. If no −−hashlimit−mode option is given, hashlimit acts like limit, but at the expensive of doing the hash housekeeping.

−−hashlimit−srcmask *prefix*

When −−hashlimit−mode srcip is used, all source addresses encountered will be grouped according to the given prefix length and the so-created subnet will be subject to hashlimit. *prefix* must be between (inclusive) 0 and 32. Note that −−hashlimit−srcmask 0 is basically doing the same thing as not specifying srcip for −−hashlimit−mode, but is technically more expensive.

−−hashlimit−dstmask *prefix*

Like −−hashlimit−srcmask, but for destination addresses.

−−hashlimit−name *foo*

The name for the /proc/net/ipt_hashlimit/foo entry.

−−hashlimit−htable−size *buckets*

The number of buckets of the hash table

−−hashlimit−htable−max *entries*

Maximum entries in the hash.

−−hashlimit−htable−expire *msec*

After how many milliseconds do hash entries expire.

−−hashlimit−htable−gcinterval *msec*

How many milliseconds between garbage collection intervals.

helper

This module matches packets related to a specific conntrack-helper.

[**!**] **−−helper** *string*

Matches packets related to the specified conntrack-helper.

string can be "ftp" for packets related to a ftp-session on default port. For other ports append −portnr to the value, ie. "ftp−2121".

Same rules apply for other conntrack-helpers.

icmp

This extension can be used if '−−protocol icmp' is specified. It provides the following option:

[**!**] **−−icmp−type** {*type*[/*code*]|*typename*}

This allows specification of the ICMP type, which can be a numeric ICMP type, type/code pair, or one of the ICMP type names shown by the command
iptables −p icmp −h

iprange

This matches on a given arbitrary range of IP addresses.

[**!**] **−−src−range** *from*[**−***to*]
Match source IP in the specified range.

[**!**] **−−dst−range** *from*[**−***to*]
Match destination IP in the specified range.

ipvs

Match IPVS connection properties.

[**!**] **−−ipvs**
packet belongs to an IPVS connection

Any of the following options implies −−ipvs (even negated)

[**!**] **−−vproto** *protocol*
VIP protocol to match; by number or name, e.g. "tcp"

[**!**] **−−vaddr** *address*[**/***mask*]
VIP address to match

[**!**] **−−vport** *port*
VIP port to match; by number or name, e.g. "http"

−−vdir {**ORIGINAL**|**REPLY**}
flow direction of packet

[**!**] **−−vmethod** {**GATE**|**IPIP**|**MASQ**}
IPVS forwarding method used

[**!**] **−−vportctl** *port*
VIP port of the controlling connection to match, e.g. 21 for FTP

length

This module matches the length of the layer-3 payload (e.g. layer-4 packet) of a packet against a specific value or range of values.

[**!**] **−−length** *length*[**:***length*]

limit

This module matches at a limited rate using a token bucket filter. A rule using this extension will match until this limit is reached (unless the '!' flag is used). It can be used in combination with the **LOG** target to give limited logging, for example.

−−limit *rate*[**/second**|**/minute**|**/hour**|**/day**]
Maximum average matching rate: specified as a number, with an optional '/second', '/minute', '/hour', or '/day' suffix; the default is 3/hour.

−−limit−burst *number*
Maximum initial number of packets to match: this number gets recharged by one every time the limit specified above is not reached, up to this number; the default is 5.

mac

[**!**] **−−mac−source** *address*
Match source MAC address. It must be of the form XX:XX:XX:XX:XX:XX. Note that this only makes sense for packets coming from an Ethernet device and entering the **PREROUTING**, **FORWARD** or **INPUT** chains.

mark

This module matches the netfilter mark field associated with a packet (which can be set using the **MARK** target below).

[**!**] **--mark** *value*[**/***mask*]

Matches packets with the given unsigned mark value (if a *mask* is specified, this is logically ANDed with the *mask* before the comparison).

multiport

This module matches a set of source or destination ports. Up to 15 ports can be specified. A port range (port:port) counts as two ports. It can only be used in conjunction with **-p tcp** or **-p udp**.

[**!**] **--source-ports**,**--sports** *port*[**,***port*|**,***port***:***port*]...

Match if the source port is one of the given ports. The flag **--sports** is a convenient alias for this option. Multiple ports or port ranges are separated using a comma, and a port range is specified using a colon. **53,1024:65535** would therefore match ports 53 and all from 1024 through 65535.

[**!**] **--destination-ports**,**--dports** *port*[**,***port*|**,***port***:***port*]...

Match if the destination port is one of the given ports. The flag **--dports** is a convenient alias for this option.

[**!**] **--ports** *port*[**,***port*|**,***port***:***port*]...

Match if either the source or destination ports are equal to one of the given ports.

osf

The osf module does passive operating system fingerprinting. This modules compares some data (Window Size, MSS, options and their order, TTL, DF, and others) from packets with the SYN bit set.

[**!**] **--genre** *string*

Match an operating system genre by using a passive fingerprinting.

--ttl *level*

Do additional TTL checks on the packet to determine the operating system. *level* can be one of the following values:

- 0 - True IP address and fingerprint TTL comparison. This generally works for LANs.
- 1 - Check if the IP header's TTL is less than the fingerprint one. Works for globally-routable addresses.
- 2 - Do not compare the TTL at all.

--log *level*

Log determined genres into dmesg even if they do not match the desired one. *level* can be one of the following values:

- 0 - Log all matched or unknown signatures
- 1 - Log only the first one
- 2 - Log all known matched signatures

You may find something like this in syslog:

Windows [2000:SP3:Windows XP Pro SP1, 2000 SP3]: 11.22.33.55:4024 -> 11.22.33.44:139 hops=3 Linux [2.5-2.6:] : 1.2.3.4:42624 -> 1.2.3.5:22 hops=4

OS fingerprints are loadable using the **nfnl_osf** program. To load fingerprints from a file, use:

nfnl_osf -f /usr/share/xtables/pf.os

To remove them again,

nfnl_osf -f /usr/share/xtables/pf.os -d

The fingerprint database can be downlaoded from http://www.openbsd.org/cgi-bin/cvsweb/src/etc/pf.os .

owner

This module attempts to match various characteristics of the packet creator, for locally generated packets. This match is only valid in the OUTPUT and POSTROUTING chains. Forwarded packets do not have any socket associated with them. Packets from kernel threads do have a socket, but usually no owner.

[**!**] **−−uid−owner** *username*

[**!**] **−−uid−owner** *userid*[−*userid*]
Matches if the packet socket's file structure (if it has one) is owned by the given user. You may also specify a numerical UID, or an UID range.

[**!**] **−−gid−owner** *groupname*

[**!**] **−−gid−owner** *groupid*[−*groupid*]
Matches if the packet socket's file structure is owned by the given group. You may also specify a numerical GID, or a GID range.

[**!**] **−−socket−exists**
Matches if the packet is associated with a socket.

physdev

This module matches on the bridge port input and output devices enslaved to a bridge device. This module is a part of the infrastructure that enables a transparent bridging IP firewall and is only useful for kernel versions above version 2.5.44.

[**!**] **−−physdev−in** *name*
Name of a bridge port via which a packet is received (only for packets entering the **INPUT**, **FORWARD** and **PREROUTING** chains). If the interface name ends in a "+", then any interface which begins with this name will match. If the packet didn't arrive through a bridge device, this packet won't match this option, unless '!' is used.

[**!**] **−−physdev−out** *name*
Name of a bridge port via which a packet is going to be sent (for packets entering the **FORWARD**, **OUTPUT** and **POSTROUTING** chains). If the interface name ends in a "+", then any interface which begins with this name will match. Note that in the **nat** and **mangle OUTPUT** chains one cannot match on the bridge output port, however one can in the **filter OUTPUT** chain. If the packet won't leave by a bridge device or if it is yet unknown what the output device will be, then the packet won't match this option, unless '!' is used.

[**!**] **−−physdev−is−in**
Matches if the packet has entered through a bridge interface.

[**!**] **−−physdev−is−out**
Matches if the packet will leave through a bridge interface.

[**!**] **−−physdev−is−bridged**
Matches if the packet is being bridged and therefore is not being routed. This is only useful in the FORWARD and POSTROUTING chains.

pkttype

This module matches the link-layer packet type.

[**!**] **−−pkt−type** {**unicast**|**broadcast**|**multicast**}

policy

This modules matches the policy used by IPsec for handling a packet.

−−dir {**in**|**out**}
Used to select whether to match the policy used for decapsulation or the policy that will be used for encapsulation. **in** is valid in the **PREROUTING, INPUT and FORWARD** chains, **out** is valid in the **POSTROUTING, OUTPUT and FORWARD** chains.

−−pol {**none**|**ipsec**}
Matches if the packet is subject to IPsec processing.

−−strict
Selects whether to match the exact policy or match if any rule of the policy matches the given policy.

[**!**] **−−reqid** *id*
Matches the reqid of the policy rule. The reqid can be specified with **setkey(8)** using **unique:id** as level.

[**!**] **−−spi** *spi*
Matches the SPI of the SA.

[**!**] **−−proto** {**ah**|**esp**|**ipcomp**}
Matches the encapsulation protocol.

[**!**] **−−mode** {**tunnel**|**transport**}
Matches the encapsulation mode.

[**!**] **−−tunnel−src** *addr*[*/mask*]
Matches the source end-point address of a tunnel mode SA. Only valid with **−−mode tunnel**.

[**!**] **−−tunnel−dst** *addr*[*/mask*]
Matches the destination end-point address of a tunnel mode SA. Only valid with **−−mode tunnel**.

−−next Start the next element in the policy specification. Can only be used with **−−strict**.

quota

Implements network quotas by decrementing a byte counter with each packet.

[**!**] **−−quota** *bytes*
The quota in bytes.

rateest

The rate estimator can match on estimated rates as collected by the RATEEST target. It supports matching on absolute bps/pps values, comparing two rate estimators and matching on the difference between two rate estimators.

−−rateest1 *name*
Name of the first rate estimator.

−−rateest2 *name*
Name of the second rate estimator (if difference is to be calculated).

−−rateest−delta
Compare difference(s) to given rate(s)

−−rateest−bps1 *value*

−−rateest−bps2 *value*
Compare bytes per second.

−−rateest−pps1 *value*

−−rateest−pps2 *value*
Compare packets per second.

[**!**] **−−rateest−lt**
Match if rate is less than given rate/estimator.

[**!**] **−−rateest−gt**
Match if rate is greater than given rate/estimator.

[**!**] **−−rateest−eq**
Match if rate is equal to given rate/estimator.

Example: This is what can be used to route outgoing data connections from an FTP server over two lines based on the available bandwidth at the time the data connection was started:

Estimate outgoing rates

iptables −t mangle −A POSTROUTING −o eth0 −j RATEEST −−rateest−name eth0 −−rateest−interval 250ms −−rateest−ewma 0.5s

iptables –t mangle –A POSTROUTING –o ppp0 –j RATEEST ––rateest–name ppp0 ––rateest–interval 250ms ––rateest–ewma 0.5s

Mark based on available bandwidth

iptables –t mangle –A balance –m conntrack ––ctstate NEW –m helper ––helper ftp –m rateest ––rateest–delta ––rateest1 eth0 ––rateest–bps1 2.5mbit ––rateest–gt ––rateest2 ppp0 ––rateest–bps2 2mbit –j CONNMARK ––set–mark 1

iptables –t mangle –A balance –m conntrack ––ctstate NEW –m helper ––helper ftp –m rateest ––rateest–delta ––rateest1 ppp0 ––rateest–bps1 2mbit ––rateest–gt ––rateest2 eth0 ––rateest–bps2 2.5mbit –j CONNMARK ––set–mark 2

iptables –t mangle –A balance –j CONNMARK ––restore–mark

realm

This matches the routing realm. Routing realms are used in complex routing setups involving dynamic routing protocols like BGP.

[**!**] **––realm** *value*[*/mask*]

Matches a given realm number (and optionally mask). If not a number, value can be a named realm from /etc/iproute2/rt_realms (mask can not be used in that case).

recent

Allows you to dynamically create a list of IP addresses and then match against that list in a few different ways.

For example, you can create a "badguy" list out of people attempting to connect to port 139 on your firewall and then DROP all future packets from them without considering them.

––set, **––rcheck**, **––update** and **––remove** are mutually exclusive.

––name *name*

Specify the list to use for the commands. If no name is given then **DEFAULT** will be used.

[**!**] **––set**

This will add the source address of the packet to the list. If the source address is already in the list, this will update the existing entry. This will always return success (or failure if **!** is passed in).

––rsource

Match/save the source address of each packet in the recent list table. This is the default.

––rdest

Match/save the destination address of each packet in the recent list table.

[**!**] **––rcheck**

Check if the source address of the packet is currently in the list.

[**!**] **––update**

Like **––rcheck**, except it will update the "last seen" timestamp if it matches.

[**!**] **––remove**

Check if the source address of the packet is currently in the list and if so that address will be removed from the list and the rule will return true. If the address is not found, false is returned.

––seconds *seconds*

This option must be used in conjunction with one of **––rcheck** or **––update**. When used, this will narrow the match to only happen when the address is in the list and was seen within the last given number of seconds.

––hitcount *hits*

This option must be used in conjunction with one of **––rcheck** or **––update**. When used, this will narrow the match to only happen when the address is in the list and packets had been received greater than or equal to the given value. This option may be used along with **––seconds** to create an even narrower match requiring a certain number of hits within a specific time frame. The

maximum value for the hitcount parameter is given by the "ip_pkt_list_tot" parameter of the xt_recent kernel module. Exceeding this value on the command line will cause the rule to be rejected.

−−rttl This option may only be used in conjunction with one of **−−rcheck** or **−−update**. When used, this will narrow the match to only happen when the address is in the list and the TTL of the current packet matches that of the packet which hit the **−−set** rule. This may be useful if you have problems with people faking their source address in order to DoS you via this module by disallowing others access to your site by sending bogus packets to you.

Examples:

iptables −A FORWARD −m recent −−name badguy −−rcheck −−seconds 60 −j DROP

iptables −A FORWARD −p tcp −i eth0 −−dport 139 −m recent −−name badguy −−set −j DROP

Steve's ipt_recent website (http://snowman.net/projects/ipt_recent/) also has some examples of usage.

/proc/net/xt_recent/* are the current lists of addresses and information about each entry of each list.

Each file in **/proc/net/xt_recent/** can be read from to see the current list or written two using the following commands to modify the list:

echo +*addr* **>/proc/net/xt_recent/DEFAULT**
to add *addr* to the DEFAULT list

echo −*addr* **>/proc/net/xt_recent/DEFAULT**
to remove *addr* from the DEFAULT list

echo / >/proc/net/xt_recent/DEFAULT
to flush the DEFAULT list (remove all entries).

The module itself accepts parameters, defaults shown:

ip_list_tot=*100*
Number of addresses remembered per table.

ip_pkt_list_tot=*20*
Number of packets per address remembered.

ip_list_hash_size=*0*
Hash table size. 0 means to calculate it based on ip_list_tot, default: 512.

ip_list_perms=*0644*
Permissions for /proc/net/xt_recent/* files.

ip_list_uid=*0*
Numerical UID for ownership of /proc/net/xt_recent/* files.

ip_list_gid=*0*
Numerical GID for ownership of /proc/net/xt_recent/* files.

sctp

[**!**] **−−source−port**,**−−sport** *port*[**:***port*]

[**!**] **−−destination−port**,**−−dport** *port*[**:***port*]

[**!**] **−−chunk−types** {**all**|**any**|**only**} *chunktype*[**:***flags*] [...]
The flag letter in upper case indicates that the flag is to match if set, in the lower case indicates to match if unset.

Chunk types: DATA INIT INIT_ACK SACK HEARTBEAT HEARTBEAT_ACK ABORT SHUTDOWN SHUTDOWN_ACK ERROR COOKIE_ECHO COOKIE_ACK ECN_ECNE ECN_CWR SHUTDOWN_COMPLETE ASCONF ASCONF_ACK FORWARD_TSN

chunk type available flags

DATA I U B E i u b e
ABORT T t
SHUTDOWN_COMPLETE T t

(lowercase means flag should be "off", uppercase means "on")

Examples:

iptables −A INPUT −p sctp −−dport 80 −j DROP

iptables −A INPUT −p sctp −−chunk−types any DATA,INIT −j DROP

iptables −A INPUT −p sctp −−chunk−types any DATA:Be −j ACCEPT

set

This module matches IP sets which can be defined by ipset(8).

[**!**] **−−match−set** *setname flag*[**,***flag*]...

where flags are the comma separated list of **src** and/or **dst** specifications and there can be no more than six of them. Hence the command

iptables −A FORWARD −m set −−match−set test src,dst

will match packets, for which (if the set type is ipportmap) the source address and destination port pair can be found in the specified set. If the set type of the specified set is single dimension (for example ipmap), then the command will match packets for which the source address can be found in the specified set.

The option **−−match−set** can be replaced by **−−set** if that does not clash with an option of other extensions.

Use of -m set requires that ipset kernel support is provided. As standard kernels do not ship this currently, the ipset or Xtables-addons package needs to be installed.

socket

This matches if an open socket can be found by doing a socket lookup on the packet.

state

This module, when combined with connection tracking, allows access to the connection tracking state for this packet.

[**!**] **−−state** *state*

Where state is a comma separated list of the connection states to match. Possible states are **INVALID** meaning that the packet could not be identified for some reason which includes running out of memory and ICMP errors which don't correspond to any known connection, **ESTABLISHED** meaning that the packet is associated with a connection which has seen packets in both directions, **NEW** meaning that the packet has started a new connection, or otherwise associated with a connection which has not seen packets in both directions, and **RELATED** meaning that the packet is starting a new connection, but is associated with an existing connection, such as an FTP data transfer, or an ICMP error. **UNTRACKED** meaning that the packet is not tracked at all, which happens if you use the NOTRACK target in raw table.

statistic

This module matches packets based on some statistic condition. It supports two distinct modes settable with the **−−mode** option.

Supported options:

−−mode *mode*

Set the matching mode of the matching rule, supported modes are **random** and **nth.**

−−probability *p*

Set the probability from 0 to 1 for a packet to be randomly matched. It works only with the **random** mode.

−−every *n*
Match one packet every nth packet. It works only with the **nth** mode (see also the **−−packet** option).

−−packet *p*
Set the initial counter value (0 <= p <= n−1, default 0) for the **nth** mode.

string

This modules matches a given string by using some pattern matching strategy. It requires a linux kernel >= 2.6.14.

−−algo {**bm**|**kmp**}
Select the pattern matching strategy. (bm = Boyer-Moore, kmp = Knuth-Pratt-Morris)

−−from *offset*
Set the offset from which it starts looking for any matching. If not passed, default is 0.

−−to *offset*
Set the offset up to which should be scanned. That is, byte *offset*-1 (counting from 0) is the last one that is scanned. If not passed, default is the packet size.

[**!**] **−−string** *pattern*
Matches the given pattern.

[**!**] **−−hex−string** *pattern*
Matches the given pattern in hex notation.

tcp

These extensions can be used if '−−protocol tcp' is specified. It provides the following options:

[**!**] **−−source−port**,**−−sport** *port*[**:***port*]
Source port or port range specification. This can either be a service name or a port number. An inclusive range can also be specified, using the format *first***:***last*. If the first port is omitted, "0" is assumed; if the last is omitted, "65535" is assumed. If the first port is greater than the second one they will be swapped. The flag **−−sport** is a convenient alias for this option.

[**!**] **−−destination−port**,**−−dport** *port*[**:***port*]
Destination port or port range specification. The flag **−−dport** is a convenient alias for this option.

[**!**] **−−tcp−flags** *mask comp*
Match when the TCP flags are as specified. The first argument *mask* is the flags which we should examine, written as a comma-separated list, and the second argument *comp* is a comma-separated list of flags which must be set. Flags are: **SYN ACK FIN RST URG PSH ALL NONE**. Hence the command
iptables −A FORWARD −p tcp −−tcp−flags SYN,ACK,FIN,RST SYN
will only match packets with the SYN flag set, and the ACK, FIN and RST flags unset.

[**!**] **−−syn**
Only match TCP packets with the SYN bit set and the ACK,RST and FIN bits cleared. Such packets are used to request TCP connection initiation; for example, blocking such packets coming in an interface will prevent incoming TCP connections, but outgoing TCP connections will be unaffected. It is equivalent to **−−tcp−flags SYN,RST,ACK,FIN SYN**. If the "!" flag precedes the "−−syn", the sense of the option is inverted.

[**!**] **−−tcp−option** *number*
Match if TCP option set.

tcpmss

This matches the TCP MSS (maximum segment size) field of the TCP header. You can only use this on TCP SYN or SYN/ACK packets, since the MSS is only negotiated during the TCP handshake at connection startup time.

[**!**] **−−mss** *value*[**:***value*]
Match a given TCP MSS value or range.

time

This matches if the packet arrival time/date is within a given range. All options are optional, but are ANDed when specified.

−−datestart *YYYY*[*−MM*[*−DD*[**T***hh*[**:***mm*[**:***ss*]]]]]

−−datestop *YYYY*[*−MM*[*−DD*[**T***hh*[**:***mm*[**:***ss*]]]]]

Only match during the given time, which must be in ISO 8601 "T" notation. The possible time range is 1970-01-01T00:00:00 to 2038-01-19T04:17:07.

If −−datestart or −−datestop are not specified, it will default to 1970-01-01 and 2038-01-19, respectively.

−−timestart *hh***:***mm*[**:***ss*]

−−timestop *hh***:***mm*[**:***ss*]

Only match during the given daytime. The possible time range is 00:00:00 to 23:59:59. Leading zeroes are allowed (e.g. "06:03") and correctly interpreted as base-10.

[**!**] **−−monthdays** *day*[**,***day*...]

Only match on the given days of the month. Possible values are **1** to **31**. Note that specifying **31** will of course not match on months which do not have a 31st day; the same goes for 28- or 29-day February.

[**!**] **−−weekdays** *day*[**,***day*...]

Only match on the given weekdays. Possible values are **Mon**, **Tue**, **Wed**, **Thu**, **Fri**, **Sat**, **Sun**, or values from **1** to **7**, respectively. You may also use two-character variants (**Mo**, **Tu**, etc.).

−−utc

Interpret the times given for **−−datestart**, **−−datestop**, **−−timestart** and **−−timestop** to be UTC.

−−localtz

Interpret the times given for **−−datestart**, **−−datestop**, **−−timestart** and **−−timestop** to be local kernel time. (Default)

EXAMPLES. To match on weekends, use:

−m time −−weekdays Sa,Su

Or, to match (once) on a national holiday block:

−m time −−datestart 2007−12−24 −−datestop 2007−12−27

Since the stop time is actually inclusive, you would need the following stop time to not match the first second of the new day:

−m time −−datestart 2007−01−01T17:00 −−datestop 2007−01−01T23:59:59

During lunch hour:

−m time −−timestart 12:30 −−timestop 13:30

The fourth Friday in the month:

−m time −−weekdays Fr −−monthdays 22,23,24,25,26,27,28

(Note that this exploits a certain mathematical property. It is not possible to say "fourth Thursday OR fourth Friday" in one rule. It is possible with multiple rules, though.)

tos

This module matches the 8-bit Type of Service field in the IPv4 header (i.e. including the "Precedence" bits) or the (also 8-bit) Priority field in the IPv6 header.

[**!**] **--tos** *value*[*/mask*]
Matches packets with the given TOS mark value. If a mask is specified, it is logically ANDed with the TOS mark before the comparison.

[**!**] **--tos** *symbol*
You can specify a symbolic name when using the tos match for IPv4. The list of recognized TOS names can be obtained by calling iptables with **-m tos -h**. Note that this implies a mask of 0x3F, i.e. all but the ECN bits.

ttl

This module matches the time to live field in the IP header.

--ttl-eq *ttl*
Matches the given TTL value.

--ttl-gt *ttl*
Matches if TTL is greater than the given TTL value.

--ttl-lt *ttl*
Matches if TTL is less than the given TTL value.

u32

U32 tests whether quantities of up to 4 bytes extracted from a packet have specified values. The specification of what to extract is general enough to find data at given offsets from tcp headers or payloads.

[**!**] **--u32** *tests*
The argument amounts to a program in a small language described below.

tests := location "=" value | tests "&&" location "=" value

value := range | value "," range

range := number | number ":" number

a single number, *n*, is interpreted the same as *n:n*. *n:m* is interpreted as the range of numbers **>=n** and **<=m**.

location := number | location operator number

operator := "&" | "<<" | ">>" | "@"

The operators **&**, **<<**, **>>** and **&&** mean the same as in C. The = is really a set membership operator and the value syntax describes a set. The @ operator is what allows moving to the next header and is described further below.

There are currently some artificial implementation limits on the size of the tests:

* no more than 10 of "**=**" (and 9 "**&&**"s) in the u32 argument
* no more than 10 ranges (and 9 commas) per value
* no more than 10 numbers (and 9 operators) per location

To describe the meaning of location, imagine the following machine that interprets it. There are three registers:

A is of type **char ***, initially the address of the IP header

B and C are unsigned 32 bit integers, initially zero

The instructions are:

number B = number;

C = (*(A+B)<<24) + (*(A+B+1)<<16) + (*(A+B+2)<<8) + *(A+B+3)

&number C = C & number

<< number C = C << number

>> number C = C >> number

@number A = A + C; then do the instruction number

Any access of memory outside [skb–>data,skb–>end] causes the match to fail. Otherwise the result of the computation is the final value of C.

Whitespace is allowed but not required in the tests. However, the characters that do occur there are likely to require shell quoting, so it is a good idea to enclose the arguments in quotes.

Example:

match IP packets with total length >= 256

The IP header contains a total length field in bytes 2-3.

--u32 "**0 & 0xFFFF = 0x100:0xFFFF**"

read bytes 0-3

AND that with 0xFFFF (giving bytes 2-3), and test whether that is in the range [0x100:0xFFFF]

Example: (more realistic, hence more complicated)

match ICMP packets with icmp type 0

First test that it is an ICMP packet, true iff byte 9 (protocol) = 1

--u32 "**6 & 0xFF = 1 &&** ...

read bytes 6-9, use **&** to throw away bytes 6-8 and compare the result to 1. Next test that it is not a fragment. (If so, it might be part of such a packet but we cannot always tell.) N.B.: This test is generally needed if you want to match anything beyond the IP header. The last 6 bits of byte 6 and all of byte 7 are 0 iff this is a complete packet (not a fragment). Alternatively, you can allow first fragments by only testing the last 5 bits of byte 6.

... **4 & 0x3FFF = 0 &&** ...

Last test: the first byte past the IP header (the type) is 0. This is where we have to use the @syntax. The length of the IP header (IHL) in 32 bit words is stored in the right half of byte 0 of the IP header itself.

... **0 >> 22 & 0x3C @ 0 >> 24 = 0**"

The first 0 means read bytes 0-3, **>>22** means shift that 22 bits to the right. Shifting 24 bits would give the first byte, so only 22 bits is four times that plus a few more bits. **&3C** then eliminates the two extra bits on the right and the first four bits of the first byte. For instance, if IHL=5, then the IP header is 20 (4 x 5) bytes long. In this case, bytes 0-1 are (in binary) xxxx0101 yyzzzzzz, **>>22** gives the 10 bit value xxxx0101yy and **&3C** gives 010100. @ means to use this number as a new offset into the packet, and read four bytes starting from there. This is the first 4 bytes of the ICMP payload, of which byte 0 is the ICMP type. Therefore, we simply shift the value 24 to the right to throw out all but the first byte and compare the result with 0.

Example:

TCP payload bytes 8-12 is any of 1, 2, 5 or 8

First we test that the packet is a tcp packet (similar to ICMP).

--u32 "**6 & 0xFF = 6 &&** ...

Next, test that it is not a fragment (same as above).

... **0 >> 22 & 0x3C @ 12 >> 26 & 0x3C @ 8 = 1,2,5,8**"

0>>22&3C as above computes the number of bytes in the IP header. @ makes this the new offset into the packet, which is the start of the TCP header. The length of the TCP header (again in 32 bit words) is the left half of byte 12 of the TCP header. The **12>>26&3C** computes this length in bytes (similar to the IP header before). "@" makes this the new offset, which is the start of the TCP payload. Finally, 8 reads bytes 8-12 of the payload and = checks whether the result is any of 1, 2, 5 or 8.

udp

These extensions can be used if '−−protocol udp' is specified. It provides the following options:

[**!**] **−−source−port**,**−−sport** *port*[**:***port*]

Source port or port range specification. See the description of the **−−source−port** option of the TCP extension for details.

[**!**] **−−destination−port**,**−−dport** *port*[**:***port*]

Destination port or port range specification. See the description of the **−−destination−port** option of the TCP extension for details.

unclean

This module takes no options, but attempts to match packets which seem malformed or unusual. This is regarded as experimental.

TARGET EXTENSIONS

iptables can use extended target modules: the following are included in the standard distribution.

CHECKSUM

This target allows to selectively work around broken/old applications. It can only be used in the mangle table.

−−checksum−fill

Compute and fill in the checksum in a packet that lacks a checksum. This is particularly useful, if you need to work around old applications such as dhcp clients, that do not work well with checksum offloads, but don't want to disable checksum offload in your device.

CLASSIFY

This module allows you to set the skb−>priority value (and thus classify the packet into a specific CBQ class).

−−set−class *major***:***minor*

Set the major and minor class value. The values are always interpreted as hexadecimal even if no 0x prefix is given.

CLUSTERIP

This module allows you to configure a simple cluster of nodes that share a certain IP and MAC address without an explicit load balancer in front of them. Connections are statically distributed between the nodes in this cluster.

−−new Create a new ClusterIP. You always have to set this on the first rule for a given ClusterIP.

−−hashmode *mode*

Specify the hashing mode. Has to be one of **sourceip**, **sourceip−sourceport**, **sourceip−sourceport−destport**.

−−clustermac *mac*

Specify the ClusterIP MAC address. Has to be a link−layer multicast address

−−total−nodes *num*

Number of total nodes within this cluster.

−−local−node *num*

Local node number within this cluster.

−−hash−init *rnd*

Specify the random seed used for hash initialization.

CONNMARK

This module sets the netfilter mark value associated with a connection. The mark is 32 bits wide.

−−set−xmark *value*[**/***mask*]

Zero out the bits given by *mask* and XOR *value* into the ctmark.

−−save−mark [**−−nfmask** *nfmask*] [**−−ctmask** *ctmask*]
Copy the packet mark (nfmark) to the connection mark (ctmark) using the given masks. The new nfmark value is determined as follows:

ctmark = (ctmark & ˜ctmask) ˆ (nfmark & nfmask)

i.e. *ctmask* defines what bits to clear and *nfmask* what bits of the nfmark to XOR into the ctmark. *ctmask* and *nfmask* default to 0xFFFFFFFF.

−−restore−mark [**−−nfmask** *nfmask*] [**−−ctmask** *ctmask*]
Copy the connection mark (ctmark) to the packet mark (nfmark) using the given masks. The new ctmark value is determined as follows:

nfmark = (nfmark & ˜*nfmask*) ˆ (ctmark & *ctmask*);

i.e. *nfmask* defines what bits to clear and *ctmask* what bits of the ctmark to XOR into the nfmark. *ctmask* and *nfmask* default to 0xFFFFFFFF.

−−restore−mark is only valid in the **mangle** table.

The following mnemonics are available for **−−set−xmark**:

−−and−mark *bits*
Binary AND the ctmark with *bits*. (Mnemonic for **−−set−xmark 0/***invbits*, where *invbits* is the binary negation of *bits*.)

−−or−mark *bits*
Binary OR the ctmark with *bits*. (Mnemonic for **−−set−xmark** *bits***/***bits*.)

−−xor−mark *bits*
Binary XOR the ctmark with *bits*. (Mnemonic for **−−set−xmark** *bits***/0**.)

−−set−mark *value*[**/***mask*]
Set the connection mark. If a mask is specified then only those bits set in the mask are modified.

−−save−mark [**−−mask** *mask*]
Copy the nfmark to the ctmark. If a mask is specified, only those bits are copied.

−−restore−mark [**−−mask** *mask*]
Copy the ctmark to the nfmark. If a mask is specified, only those bits are copied. This is only valid in the **mangle** table.

CONNSECMARK

This module copies security markings from packets to connections (if unlabeled), and from connections back to packets (also only if unlabeled). Typically used in conjunction with SECMARK, it is only valid in the **mangle** table.

−−save If the packet has a security marking, copy it to the connection if the connection is not marked.

−−restore
If the packet does not have a security marking, and the connection does, copy the security marking from the connection to the packet.

CT

The CT target allows to set parameters for a packet or its associated connection. The target attaches a "template" connection tracking entry to the packet, which is then used by the conntrack core when initializing a new ct entry. This target is thus only valid in the "raw" table.

−−notrack
Disables connection tracking for this packet.

−−helper *name*
Use the helper identified by *name* for the connection. This is more flexible than loading the conntrack helper modules with preset ports.

−−ctevents *event*[**,**...]
Only generate the specified conntrack events for this connection. Possible event types are: **new**, **related**, **destroy**, **reply**, **assured**, **protoinfo**, **helper**, **mark** (this refers to the ctmark, not nfmark), **natseqinfo**, **secmark** (ctsecmark).

−−expevents *event*[**,**...]
Only generate the specified expectation events for this connection. Possible event types are: **new**.

−−zone *id*
Assign this packet to zone *id* and only have lookups done in that zone. By default, packets have zone 0.

DNAT

This target is only valid in the **nat** table, in the **PREROUTING** and **OUTPUT** chains, and user-defined chains which are only called from those chains. It specifies that the destination address of the packet should be modified (and all future packets in this connection will also be mangled), and rules should cease being examined. It takes one type of option:

−−to−destination [*ipaddr*][−*ipaddr*][**:***port*[−*port*]]
which can specify a single new destination IP address, an inclusive range of IP addresses, and optionally, a port range (which is only valid if the rule also specifies **−p tcp** or **−p udp**). If no port range is specified, then the destination port will never be modified. If no IP address is specified then only the destination port will be modified.

In Kernels up to 2.6.10 you can add several −−to−destination options. For those kernels, if you specify more than one destination address, either via an address range or multiple −−to−destination options, a simple round-robin (one after another in cycle) load balancing takes place between these addresses. Later Kernels (>= 2.6.11-rc1) don't have the ability to NAT to multiple ranges anymore.

−−random
If option **−−random** is used then port mapping will be randomized (kernel >= 2.6.22).

−−persistent
Gives a client the same source-/destination-address for each connection. This supersedes the SAME target. Support for persistent mappings is available from 2.6.29-rc2.

DSCP

This target allows to alter the value of the DSCP bits within the TOS header of the IPv4 packet. As this manipulates a packet, it can only be used in the mangle table.

−−set−dscp *value*
Set the DSCP field to a numerical value (can be decimal or hex)

−−set−dscp−class *class*
Set the DSCP field to a DiffServ class.

ECN

This target allows to selectively work around known ECN blackholes. It can only be used in the mangle table.

−−ecn−tcp−remove
Remove all ECN bits from the TCP header. Of course, it can only be used in conjunction with **−p tcp**.

IDLETIMER

This target can be used to identify when interfaces have been idle for a certain period of time. Timers are identified by labels and are created when a rule is set with a new label. The rules also take a timeout value (in seconds) as an option. If more than one rule uses the same timer label, the timer will be restarted whenever any of the rules get a hit. One entry for each timer is created in sysfs. This attribute contains the timer remaining for the timer to expire. The attributes are located under the xt_idletimer class:

/sys/class/xt_idletimer/timers/<label>

When the timer expires, the target module sends a sysfs notification to the userspace, which can then decide what to do (eg. disconnect to save power).

−−timeout *amount*
: This is the time in seconds that will trigger the notification.

−−label *string*
: This is a unique identifier for the timer. The maximum length for the label string is 27 characters.

LOG

Turn on kernel logging of matching packets. When this option is set for a rule, the Linux kernel will print some information on all matching packets (like most IP header fields) via the kernel log (where it can be read with *dmesg* or *syslogd*(8)). This is a "non-terminating target", i.e. rule traversal continues at the next rule. So if you want to LOG the packets you refuse, use two separate rules with the same matching criteria, first using target LOG then DROP (or REJECT).

−−log−level *level*
: Level of logging (numeric or see *syslog.conf*(5)).

−−log−prefix *prefix*
: Prefix log messages with the specified prefix; up to 29 letters long, and useful for distinguishing messages in the logs.

−−log−tcp−sequence
: Log TCP sequence numbers. This is a security risk if the log is readable by users.

−−log−tcp−options
: Log options from the TCP packet header.

−−log−ip−options
: Log options from the IP packet header.

−−log−uid
: Log the userid of the process which generated the packet.

MARK

This target is used to set the Netfilter mark value associated with the packet. It can, for example, be used in conjunction with routing based on fwmark (needs iproute2). If you plan on doing so, note that the mark needs to be set in the PREROUTING chain of the mangle table to affect routing. The mark field is 32 bits wide.

−−set−xmark *value*[**/***mask*]
: Zeroes out the bits given by *mask* and XORs *value* into the packet mark ("nfmark"). If *mask* is omitted, 0xFFFFFFFF is assumed.

−−set−mark *value*[**/***mask*]
: Zeroes out the bits given by *mask* and ORs *value* into the packet mark. If *mask* is omitted, 0xFFFFFFFF is assumed.

The following mnemonics are available:

−−and−mark *bits*
: Binary AND the nfmark with *bits*. (Mnemonic for **−−set−xmark 0/***invbits*, where *invbits* is the binary negation of *bits*.)

−−or−mark *bits*
: Binary OR the nfmark with *bits*. (Mnemonic for **−−set−xmark** *bits***/***bits*.)

−−xor−mark *bits*
: Binary XOR the nfmark with *bits*. (Mnemonic for **−−set−xmark** *bits***/0**.)

MASQUERADE

This target is only valid in the **nat** table, in the **POSTROUTING** chain. It should only be used with dynamically assigned IP (dialup) connections: if you have a static IP address, you should use the SNAT target. Masquerading is equivalent to specifying a mapping to the IP address of the interface the packet is going out, but also has the effect that connections are *forgotten* when the interface goes down. This is the correct behavior when the next dialup is unlikely to have the same interface address (and hence any established connections are lost anyway). It takes one option:

−−to−ports *port*[−*port*]
: This specifies a range of source ports to use, overriding the default **SNAT** source port-selection heuristics (see above). This is only valid if the rule also specifies **−p tcp** or **−p udp**.

−−random
: Randomize source port mapping If option **−−random** is used then port mapping will be randomized (kernel >= 2.6.21).

MIRROR

This is an experimental demonstration target which inverts the source and destination fields in the IP header and retransmits the packet. It is only valid in the **INPUT**, **FORWARD** and **PREROUTING** chains, and user-defined chains which are only called from those chains. Note that the outgoing packets are **NOT** seen by any packet filtering chains, connection tracking or NAT, to avoid loops and other problems.

NETMAP

This target allows you to statically map a whole network of addresses onto another network of addresses. It can only be used from rules in the **nat** table.

−−to *address*[*/mask*]
: Network address to map to. The resulting address will be constructed in the following way: All 'one' bits in the mask are filled in from the new 'address'. All bits that are zero in the mask are filled in from the original address.

NFLOG

This target provides logging of matching packets. When this target is set for a rule, the Linux kernel will pass the packet to the loaded logging backend to log the packet. This is usually used in combination with nfnetlink_log as logging backend, which will multicast the packet through a *netlink* socket to the specified multicast group. One or more userspace processes may subscribe to the group to receive the packets. Like LOG, this is a non-terminating target, i.e. rule traversal continues at the next rule.

−−nflog−group *nlgroup*
: The netlink group (1 – 2^32−1) to which packets are (only applicable for nfnetlink_log). The default value is 0.

−−nflog−prefix *prefix*
: A prefix string to include in the log message, up to 64 characters long, useful for distinguishing messages in the logs.

−−nflog−range *size*
: The number of bytes to be copied to userspace (only applicable for nfnetlink_log). nfnetlink_log instances may specify their own range, this option overrides it.

−−nflog−threshold *size*
: Number of packets to queue inside the kernel before sending them to userspace (only applicable for nfnetlink_log). Higher values result in less overhead per packet, but increase delay until the packets reach userspace. The default value is 1.

NFQUEUE

This target is an extension of the QUEUE target. As opposed to QUEUE, it allows you to put a packet into any specific queue, identified by its 16-bit queue number. It can only be used with Kernel versions 2.6.14 or later, since it requires the **nfnetlink_queue** kernel support. The **queue-balance** option was added in Linux 2.6.31.

−−queue−num *value*
This specifies the QUEUE number to use. Valid queue numbers are 0 to 65535. The default value is 0.

−−queue−balance *value***:***value*
This specifies a range of queues to use. Packets are then balanced across the given queues. This is useful for multicore systems: start multiple instances of the userspace program on queues x, x+1, .. x+n and use "−−queue−balance *x***:***x+n*". Packets belonging to the same connection are put into the same nfqueue.

NOTRACK
This target disables connection tracking for all packets matching that rule.

It can only be used in the **raw** table.

RATEEST
The RATEEST target collects statistics, performs rate estimation calculation and saves the results for later evaluation using the **rateest** match.

−−rateest−name *name*
Count matched packets into the pool referred to by *name*, which is freely choosable.

−−rateest−interval *amount*{**s**|**ms**|**us**}
Rate measurement interval, in seconds, milliseconds or microseconds.

−−rateest−ewmalog *value*
Rate measurement averaging time constant.

REDIRECT
This target is only valid in the **nat** table, in the **PREROUTING** and **OUTPUT** chains, and user-defined chains which are only called from those chains. It redirects the packet to the machine itself by changing the destination IP to the primary address of the incoming interface (locally-generated packets are mapped to the 127.0.0.1 address).

−−to−ports *port*[−*port*]
This specifies a destination port or range of ports to use: without this, the destination port is never altered. This is only valid if the rule also specifies **−p tcp** or **−p udp**.

−−random
If option **−−random** is used then port mapping will be randomized (kernel >= 2.6.22).

REJECT
This is used to send back an error packet in response to the matched packet: otherwise it is equivalent to **DROP** so it is a terminating TARGET, ending rule traversal. This target is only valid in the **INPUT**, **FORWARD** and **OUTPUT** chains, and user-defined chains which are only called from those chains. The following option controls the nature of the error packet returned:

−−reject−with *type*
The type given can be **icmp−net−unreachable**, **icmp−host−unreachable**, **icmp−port−unreachable**, **icmp−proto−unreachable**, **icmp−net−prohibited**, **icmp−host−prohibited** or **icmp−admin−prohibited** (*) which return the appropriate ICMP error message (**port−unreachable** is the default). The option **tcp−reset** can be used on rules which only match the TCP protocol: this causes a TCP RST packet to be sent back. This is mainly useful for blocking *ident* (113/tcp) probes which frequently occur when sending mail to broken mail hosts (which won't accept your mail otherwise).

(*) Using icmp−admin−prohibited with kernels that do not support it will result in a plain DROP instead of REJECT

SAME
Similar to SNAT/DNAT depending on chain: it takes a range of addresses ('−−to 1.2.3.4−1.2.3.7') and gives a client the same source-/destination-address for each connection.

N.B.: The DNAT target's **−−persistent** option replaced the SAME target.

−−to *ipaddr*[*−ipaddr*]
Addresses to map source to. May be specified more than once for multiple ranges.

−−nodst
Don't use the destination-ip in the calculations when selecting the new source-ip

−−random
Port mapping will be forcibly randomized to avoid attacks based on port prediction (kernel >= 2.6.21).

SECMARK

This is used to set the security mark value associated with the packet for use by security subsystems such as SELinux. It is only valid in the **mangle** table. The mark is 32 bits wide.

−−selctx *security_context*

SET

This modules adds and/or deletes entries from IP sets which can be defined by ipset(8).

−−add−set *setname flag*[**,***flag*...]
add the address(es)/port(s) of the packet to the sets

−−del−set *setname flag*[**,***flag*...]
delete the address(es)/port(s) of the packet from the sets

where flags are **src** and/or **dst** specifications and there can be no more than six of them.

Use of -j SET requires that ipset kernel support is provided. As standard kernels do not ship this currently, the ipset or Xtables-addons package needs to be installed.

SNAT

This target is only valid in the **nat** table, in the **POSTROUTING** chain. It specifies that the source address of the packet should be modified (and all future packets in this connection will also be mangled), and rules should cease being examined. It takes one type of option:

−−to−source *ipaddr*[*−ipaddr*][**:***port*[*−port*]]
which can specify a single new source IP address, an inclusive range of IP addresses, and optionally, a port range (which is only valid if the rule also specifies **−p tcp** or **−p udp**). If no port range is specified, then source ports below 512 will be mapped to other ports below 512: those between 512 and 1023 inclusive will be mapped to ports below 1024, and other ports will be mapped to 1024 or above. Where possible, no port alteration will

In Kernels up to 2.6.10, you can add several −−to−source options. For those kernels, if you specify more than one source address, either via an address range or multiple −−to−source options, a simple round-robin (one after another in cycle) takes place between these addresses. Later Kernels (>= 2.6.11-rc1) don't have the ability to NAT to multiple ranges anymore.

−−random
If option **−−random** is used then port mapping will be randomized (kernel >= 2.6.21).

−−persistent
Gives a client the same source-/destination-address for each connection. This supersedes the SAME target. Support for persistent mappings is available from 2.6.29-rc2.

TCPMSS

This target allows to alter the MSS value of TCP SYN packets, to control the maximum size for that connection (usually limiting it to your outgoing interface's MTU minus 40 for IPv4 or 60 for IPv6, respectively). Of course, it can only be used in conjunction with **−p tcp**.

This target is used to overcome criminally braindead ISPs or servers which block "ICMP Fragmentation Needed" or "ICMPv6 Packet Too Big" packets. The symptoms of this problem are that everything works fine from your Linux firewall/router, but machines behind it can never exchange large packets:

1) Web browsers connect, then hang with no data received.
2) Small mail works fine, but large emails hang.
3) ssh works fine, but scp hangs after initial handshaking.

Workaround: activate this option and add a rule to your firewall configuration like:

```
iptables −t mangle −A FORWARD −p tcp −−tcp−flags SYN,RST SYN
        −j TCPMSS −−clamp−mss−to−pmtu
```

−−set−mss *value*
: Explicitly sets MSS option to specified value. If the MSS of the packet is already lower than *value*, it will **not** be increased (from Linux 2.6.25 onwards) to avoid more problems with hosts relying on a proper MSS.

−−clamp−mss−to−pmtu
: Automatically clamp MSS value to (path_MTU − 40 for IPv4; −60 for IPv6). This may not function as desired where asymmetric routes with differing path MTU exist — the kernel uses the path MTU which it would use to send packets from itself to the source and destination IP addresses. Prior to Linux 2.6.25, only the path MTU to the destination IP address was considered by this option; subsequent kernels also consider the path MTU to the source IP address.

These options are mutually exclusive.

TCPOPTSTRIP

This target will strip TCP options off a TCP packet. (It will actually replace them by NO-OPs.) As such, you will need to add the **−p tcp** parameters.

−−strip−options *option*[**,***option...*]
: Strip the given option(s). The options may be specified by TCP option number or by symbolic name. The list of recognized options can be obtained by calling iptables with **−j TCPOPTSTRIP −h**.

TEE

The **TEE** target will clone a packet and redirect this clone to another machine on the **local** network segment. In other words, the nexthop must be the target, or you will have to configure the nexthop to forward it further if so desired.

−−gateway *ipaddr*
: Send the cloned packet to the host reachable at the given IP address. Use of 0.0.0.0 (for IPv4 packets) or :: (IPv6) is invalid.

To forward all incoming traffic on eth0 to an Network Layer logging box:

−t mangle −A PREROUTING −i eth0 −j TEE −−gateway 2001:db8::1

TOS

This module sets the Type of Service field in the IPv4 header (including the "precedence" bits) or the Priority field in the IPv6 header. Note that TOS shares the same bits as DSCP and ECN. The TOS target is only valid in the **mangle** table.

−−set−tos *value*[**/***mask*]
: Zeroes out the bits given by *mask* and XORs *value* into the TOS/Priority field. If *mask* is omitted, 0xFF is assumed.

−−set−tos *symbol*
: You can specify a symbolic name when using the TOS target for IPv4. It implies a mask of 0xFF. The list of recognized TOS names can be obtained by calling iptables with **−j TOS −h**.

The following mnemonics are available:

−−and−tos *bits*
: Binary AND the TOS value with *bits*. (Mnemonic for **−−set−tos 0/***invbits*, where *invbits* is the binary negation of *bits*.)

−−or−tos *bits*
Binary OR the TOS value with *bits*. (Mnemonic for **−−set−tos** *bits***/***bits*.)

−−xor−tos *bits*
Binary XOR the TOS value with *bits*. (Mnemonic for **−−set−tos** *bits***/0**.)

TPROXY
This target is only valid in the **mangle** table, in the **PREROUTING** chain and user-defined chains which are only called from this chain. It redirects the packet to a local socket without changing the packet header in any way. It can also change the mark value which can then be used in advanced routing rules. It takes three options:

−−on−port *port*
This specifies a destination port to use. It is a required option, 0 means the new destination port is the same as the original. This is only valid if the rule also specifies **−p tcp** or **−p udp**.

−−on−ip *address*
This specifies a destination address to use. By default the address is the IP address of the incoming interface. This is only valid if the rule also specifies **−p tcp** or **−p udp**.

−−tproxy−mark *value*[**/***mask*]
Marks packets with the given value/mask. The fwmark value set here can be used by advanced routing. (Required for transparent proxying to work: otherwise these packets will get forwarded, which is probably not what you want.)

TRACE
This target marks packes so that the kernel will log every rule which match the packets as those traverse the tables, chains, rules. (The ipt_LOG or ip6t_LOG module is required for the logging.) The packets are logged with the string prefix: "TRACE: tablename:chainname:type:rulenum " where type can be "rule" for plain rule, "return" for implicit rule at the end of a user defined chain and "policy" for the policy of the built in chains.
It can only be used in the **raw** table.

TTL
This is used to modify the IPv4 TTL header field. The TTL field determines how many hops (routers) a packet can traverse until it's time to live is exceeded.

Setting or incrementing the TTL field can potentially be very dangerous, so it should be avoided at any cost.

Don't ever set or increment the value on packets that leave your local network! mangle table.

−−ttl−set *value*
Set the TTL value to ‘value’.

−−ttl−dec *value*
Decrement the TTL value ‘value’ times.

−−ttl−inc *value*
Increment the TTL value ‘value’ times.

ULOG
This target provides userspace logging of matching packets. When this target is set for a rule, the Linux kernel will multicast this packet through a *netlink* socket. One or more userspace processes may then subscribe to various multicast groups and receive the packets. Like LOG, this is a "non-terminating target", i.e. rule traversal continues at the next rule.

−−ulog−nlgroup *nlgroup*
This specifies the netlink group (1-32) to which the packet is sent. Default value is 1.

−−ulog−prefix *prefix*
Prefix log messages with the specified prefix; up to 32 characters long, and useful for distinguishing messages in the logs.

−−ulog−cprange *size*
Number of bytes to be copied to userspace. A value of 0 always copies the entire packet, regardless of its size. Default is 0.

−−ulog−qthreshold *size*
Number of packet to queue inside kernel. Setting this value to, e.g. 10 accumulates ten packets inside the kernel and transmits them as one netlink multipart message to userspace. Default is 1 (for backwards compatibility).

DIAGNOSTICS

Various error messages are printed to standard error. The exit code is 0 for correct functioning. Errors which appear to be caused by invalid or abused command line parameters cause an exit code of 2, and other errors cause an exit code of 1.

BUGS

Bugs? What's this? ;-) Well, you might want to have a look at http://bugzilla.netfilter.org/

COMPATIBILITY WITH IPCHAINS

This **iptables** is very similar to ipchains by Rusty Russell. The main difference is that the chains **INPUT** and **OUTPUT** are only traversed for packets coming into the local host and originating from the local host respectively. Hence every packet only passes through one of the three chains (except loopback traffic, which involves both INPUT and OUTPUT chains); previously a forwarded packet would pass through all three.

The other main difference is that **−i** refers to the input interface; **−o** refers to the output interface, and both are available for packets entering the **FORWARD** chain.

The various forms of NAT have been separated out; **iptables** is a pure packet filter when using the default 'filter' table, with optional extension modules. This should simplify much of the previous confusion over the combination of IP masquerading and packet filtering seen previously. So the following options are handled differently:
−j MASQ
−M −S
−M −L
There are several other changes in iptables.

SEE ALSO

iptables−save(8), **iptables−restore**(8), **ip6tables**(8), **ip6tables−save**(8), **ip6tables−restore**(8), **libipq**(3).

The packet-filtering-HOWTO details iptables usage for packet filtering, the NAT-HOWTO details NAT, the netfilter-extensions-HOWTO details the extensions that are not in the standard distribution, and the netfilter-hacking-HOWTO details the netfilter internals.
See **http://www.netfilter.org/**.

AUTHORS

Rusty Russell originally wrote iptables, in early consultation with Michael Neuling.

Marc Boucher made Rusty abandon ipnatctl by lobbying for a generic packet selection framework in iptables, then wrote the mangle table, the owner match, the mark stuff, and ran around doing cool stuff everywhere.

James Morris wrote the TOS target, and tos match.

Jozsef Kadlecsik wrote the REJECT target.

Harald Welte wrote the ULOG and NFQUEUE target, the new libiptc, as well as the TTL, DSCP, ECN matches and targets.

The Netfilter Core Team is: Marc Boucher, Martin Josefsson, Yasuyuki Kozakai, Jozsef Kadlecsik, Patrick McHardy, James Morris, Pablo Neira Ayuso, Harald Welte and Rusty Russell.

Man page originally written by Herve Eychenne <rv@wallfire.org>.

NAME

iptables-save — dump iptables rules to stdout

SYNOPSIS

iptables–save [**–M** *modprobe*] [**–c**] [**–t** *table*]

DESCRIPTION

iptables-save is used to dump the contents of an IP Table in easily parseable format to STDOUT. Use I/O-redirection provided by your shell to write to a file.

–M *modprobe_program*
Specify the path to the modprobe program. By default, iptables-save will inspect /proc/sys/kernel/modprobe to determine the executable's path.

–c, **––counters**
include the current values of all packet and byte counters in the output

–t, **––table** *tablename*
restrict output to only one table. If not specified, output includes all available tables.

BUGS

None known as of iptables-1.2.1 release

AUTHOR

Harald Welte <laforge@gnumonks.org>

SEE ALSO

iptables–restore(8), **iptables**(8)

The iptables-HOWTO, which details more iptables usage, the NAT-HOWTO, which details NAT, and the netfilter-hacking-HOWTO which details the internals.

NAME

iptables-restore — Restore IP Tables

SYNOPSIS

iptables−restore [**−c**] [**−n**]

DESCRIPTION

iptables-restore is used to restore IP Tables from data specified on STDIN. Use I/O redirection provided by your shell to read from a file

−c, **−−counters**
restore the values of all packet and byte counters

−n, **−−noflush**
don't flush the previous contents of the table. If not specified, **iptables-restore** flushes (deletes) all previous contents of the respective IP Table.

BUGS

None known as of iptables-1.2.1 release

AUTHOR

Harald Welte <laforge@gnumonks.org>

SEE ALSO

iptables−save(8), **iptables**(8)

The iptables-HOWTO, which details more iptables usage, the NAT-HOWTO, which details NAT, and the netfilter-hacking-HOWTO which details the internals.