

NAME

ipfw — IP firewall and traffic shaper control program

SYNOPSIS

```

ipfw [-cq] add rule
ipfw [-acdefnNstT] {list | show} [rule | first-last ...]
ipfw [-f | -q] flush
ipfw [-q] {delete | zero | resetlog} [set] [number ...]
ipfw enable {firewall | altq | one_pass | debug | verbose | dyn_keepalive}
ipfw disable {firewall | altq | one_pass | debug | verbose | dyn_keepalive}

ipfw set [disable number ...] [enable number ...]
ipfw set move [rule] number to number
ipfw set swap number number
ipfw set show

ipfw table number add addr[/masklen] [value]
ipfw table number delete addr[/masklen]
ipfw table number flush
ipfw table number list

ipfw {pipe | queue} number config config-options
ipfw [-s [field]] {pipe | queue} {delete | list | show} [number ...]

ipfw [-cfnNqS] [-p preproc [preproc-flags]] pathname

```

DESCRIPTION

The **ipfw** utility is the user interface for controlling the **ipfw(4)** firewall and the **dummynet(4)** traffic shaper in FreeBSD.

An **ipfw** configuration, or *ruleset*, is made of a list of *rules* numbered from 1 to 65535. Packets are passed to **ipfw** from a number of different places in the protocol stack (depending on the source and destination of the packet, it is possible that **ipfw** is invoked multiple times on the same packet). The packet passed to the firewall is compared against each of the rules in the firewall *ruleset*. When a match is found, the action corresponding to the matching rule is performed.

Depending on the action and certain system settings, packets can be reinjected into the firewall at some rule after the matching one for further processing.

An **ipfw** ruleset always includes a *default* rule (numbered 65535) which cannot be modified or deleted, and matches all packets. The action associated with the *default* rule can be either **deny** or **allow** depending on how the kernel is configured.

If the ruleset includes one or more rules with the **keep-state** or **limit** option, then **ipfw** assumes a *stateful* behaviour, i.e., upon a match it will create dynamic rules matching the exact parameters (addresses and ports) of the matching packet.

These dynamic rules, which have a limited lifetime, are checked at the first occurrence of a **check-state**, **keep-state** or **limit** rule, and are typically used to open the firewall on-demand to legitimate traffic only. See the **STATEFUL FIREWALL** and **EXAMPLES** Sections below for more information on the stateful behaviour of **ipfw**.

All rules (including dynamic ones) have a few associated counters: a packet count, a byte count, a log count and a timestamp indicating the time of the last match. Counters can be displayed or reset with **ipfw** commands.

Rules can be added with the **add** command; deleted individually or in groups with the **delete** command, and globally (except those in set 31) with the **flush** command; displayed, optionally with the content of the counters, using the **show** and **list** commands. Finally, counters can be reset with the **zero** and **resetlog** commands.

Also, each rule belongs to one of 32 different *sets*, and there are **ipfw** commands to atomically manipulate sets, such as enable, disable, swap sets, move all rules in a set to another one, delete all rules in a set. These can be useful to install temporary configurations, or to test them. See Section **SETS OF RULES** for more information on *sets*.

The following options are available:

- a** While listing, show counter values. The **show** command just implies this option.
- b** Only show the action and the comment, not the body of a rule. Implies **-c**.
- c** When entering or showing rules, print them in compact form, i.e., without the optional "ip from any to any" string when this does not carry any additional information.
- d** While listing, show dynamic rules in addition to static ones.
- e** While listing, if the **-d** option was specified, also show expired dynamic rules.
- f** Do not ask for confirmation for commands that can cause problems if misused, i.e. **flush**. If there is no tty associated with the process, this is implied.
- n** Only check syntax of the command strings, without actually passing them to the kernel.
- N** Try to resolve addresses and service names in output.
- q** While **adding**, **zeroing**, **resetlogging** or **flushing**, be quiet about actions (implies **-f**). This is useful for adjusting rules by executing multiple **ipfw** commands in a script (e.g., `sh /etc/rc.firewall`), or by processing a file of many **ipfw** rules across a remote login session. If a **flush** is performed in normal (verbose) mode (with the default kernel configuration), it prints a message. Because all rules are flushed, the message might not be delivered to the login session, causing the remote login session to be closed and the remainder of the ruleset to not be processed. Access to the console would then be required to recover.
- s** While listing rules, show the *set* each rule belongs to. If this flag is not specified, disabled rules will not be listed.
- s [field]** While listing pipes, sort according to one of the four counters (total or current packets or bytes).
- t** While listing, show last match timestamp (converted with `ctime()`).
- T** While listing, show last match timestamp (as seconds from the epoch). This form can be more convenient for postprocessing by scripts.

To ease configuration, rules can be put into a file which is processed using **ipfw** as shown in the last synopsis line. An absolute *pathname* must be used. The file will be read line by line and applied as arguments to the **ipfw** utility.

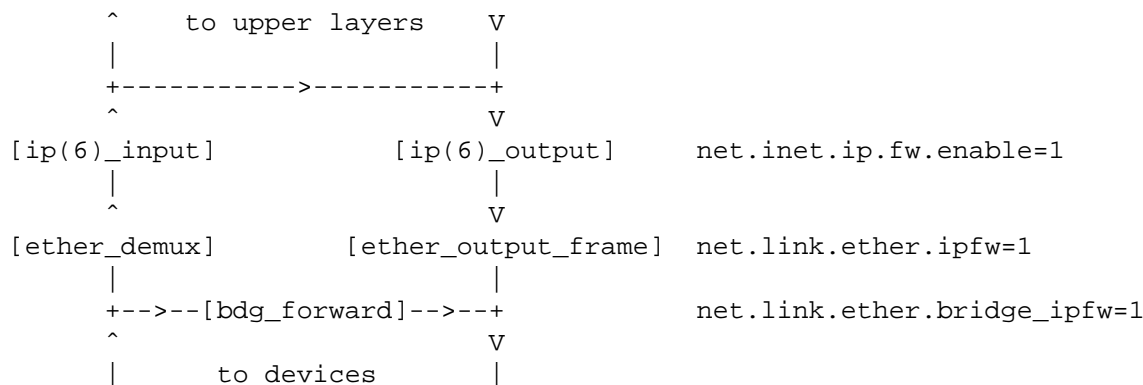
Optionally, a preprocessor can be specified using **-p preproc** where *pathname* is to be piped through. Useful preprocessors include `cpp(1)` and `m4(1)`. If *preproc* does not start with a slash (`'/'`) as its first character, the usual `PATH` name search is performed. Care should be taken with this in environments where not all file systems are mounted (yet) by the time **ipfw** is being run (e.g. when they are mounted over NFS). Once **-p** has been specified, any additional arguments are passed on to the preprocessor for interpretation. This allows for flexible configuration files (like conditionalizing them on the local hostname) and the use of macros to centralize frequently required arguments like IP addresses.

The **ipfw pipe** and **queue** commands are used to configure the traffic shaper, as shown in the **TRAFFIC SHAPER (DUMMYNET) CONFIGURATION** Section below.

If the world and the kernel get out of sync the **ipfw** ABI may break, preventing you from being able to add any rules. This can adversely effect the booting process. You can use **ipfw disable firewall** to temporarily disable the firewall to regain access to the network, allowing you to fix the problem.

PACKET FLOW

A packet is checked against the active ruleset in multiple places in the protocol stack, under control of several sysctl variables. These places and variables are shown below, and it is important to have this picture in mind in order to design a correct ruleset.



As can be noted from the above picture, the number of times the same packet goes through the firewall can vary between 0 and 4 depending on packet source and destination, and system configuration.

Note that as packets flow through the stack, headers can be stripped or added to it, and so they may or may not be available for inspection. E.g., incoming packets will include the MAC header when **ipfw** is invoked from **ether_demux()**, but the same packets will have the MAC header stripped off when **ipfw** is invoked from **ip_input()** or **ip6_input()**.

Also note that each packet is always checked against the complete ruleset, irrespective of the place where the check occurs, or the source of the packet. If a rule contains some match patterns or actions which are not valid for the place of invocation (e.g. trying to match a MAC header within **ip_input** or **ip6_input**), the match pattern will not match, but a **not** operator in front of such patterns *will* cause the pattern to *always* match on those packets. It is thus the responsibility of the programmer, if necessary, to write a suitable rule-set to differentiate among the possible places. **skipto** rules can be useful here, as an example:

```

# packets from ether_demux or bdg_forward
ipfw add 10 skipto 1000 all from any to any layer2 in
# packets from ip_input
ipfw add 10 skipto 2000 all from any to any not layer2 in
# packets from ip_output
ipfw add 10 skipto 3000 all from any to any not layer2 out
# packets from ether_output_frame
ipfw add 10 skipto 4000 all from any to any layer2 out

```

(yes, at the moment there is no way to differentiate between **ether_demux** and **bdg_forward**).

SYNTAX

In general, each keyword or argument must be provided as a separate command line argument, with no leading or trailing spaces. Keywords are case-sensitive, whereas arguments may or may not be case-sensitive depending on their nature (e.g. uid's are, hostnames are not).

In **ipfw2** you can introduce spaces after commas ',' to make the line more readable. You can also put the entire command (including flags) into a single argument. E.g., the following forms are equivalent:

```
ipfw -q add deny src-ip 10.0.0.0/24,127.0.0.1/8
ipfw -q add deny src-ip 10.0.0.0/24, 127.0.0.1/8
ipfw "-q add deny src-ip 10.0.0.0/24, 127.0.0.1/8"
```

RULE FORMAT

The format of **ipfw** rules is the following:

```
[rule_number] [set set_number] [prob match_probability]
  action [log [logamount number]] [altq queue] body
```

where the body of the rule specifies which information is used for filtering packets, among the following:

Layer-2 header fields	When available
IPv4 and IPv6 Protocol	TCP, UDP, ICMP, etc.
Source and dest. addresses and ports	
Direction	See Section PACKET FLOW
Transmit and receive interface	By name or address
Misc. IP header fields	Version, type of service, datagram length, identification, fragment flag (non-zero IP offset), Time To Live
IP options	
IPv6 Extension headers	Fragmentation, Hop-by-Hop options, source routing, IPsec options.
IPv6 Flow-ID	
Misc. TCP header fields	TCP flags (SYN, FIN, ACK, RST, etc.), sequence number, acknowledgment number, window
TCP options	
ICMP types	for ICMP packets
ICMP6 types	for ICMP6 packets
User/group ID	When the packet can be associated with a local socket.
Divert status	Whether a packet came from a divert socket (e.g., <code>natd(8)</code>).

Note that some of the above information, e.g. source MAC or IP addresses and TCP/UDP ports, could easily be spoofed, so filtering on those fields alone might not guarantee the desired results.

rule_number

Each rule is associated with a *rule_number* in the range 1..65535, with the latter reserved for the *default* rule. Rules are checked sequentially by rule number. Multiple rules can have the same number, in which case they are checked (and listed) according to the order in which they have been added. If a rule is entered without specifying a number, the kernel will assign one in such a way that the rule becomes the last one before the *default* rule. Automatic rule numbers are assigned by incrementing the last non-default rule number by the value of the `sysctl` variable `net.inet.ip.fw.autoinc_step` which defaults to 100. If this is not possible (e.g. because we would go beyond the maximum allowed rule number), the number of the last non-default value is used instead.

set *set_number*

Each rule is associated with a *set_number* in the range 0..31. Sets can be individually disabled and enabled, so this parameter is of fundamental importance for atomic ruleset manipulation. It

can be also used to simplify deletion of groups of rules. If a rule is entered without specifying a set number, set 0 will be used.

Set 31 is special in that it cannot be disabled, and rules in set 31 are not deleted by the **ipfw flush** command (but you can delete them with the **ipfw delete set 31** command). Set 31 is also used for the *default* rule.

prob *match_probability*

A match is only declared with the specified probability (floating point number between 0 and 1). This can be useful for a number of applications such as random packet drop or (in conjunction with `dummy`(4)) to simulate the effect of multiple paths leading to out-of-order packet delivery.

Note: this condition is checked before any other condition, including ones such as `keep-state` or `check-state` which might have side effects.

log [**logamount** *number*]

When a packet matches a rule with the **log** keyword, a message will be logged to `syslogd`(8) with a `LOG_SECURITY` facility. The logging only occurs if the `sysctl` variable `net.inet.ipfw.verbose` is set to 1 (which is the default when the kernel is compiled with `IPFW_VERBOSE`) and the number of packets logged so far for that particular rule does not exceed the **logamount** parameter. If no **logamount** is specified, the limit is taken from the `sysctl` variable `net.inet.ipfw.verbose_limit`. In both cases, a value of 0 removes the logging limit.

Once the limit is reached, logging can be re-enabled by clearing the logging counter or the packet counter for that entry, see the **resetlog** command.

Note: logging is done after all other packet matching conditions have been successfully verified, and before performing the final action (accept, deny, etc.) on the packet.

altq *queue*

When a packet matches a rule with the **altq** keyword, the ALTQ identifier for the given *queue* (see `altq`(4)) will be attached. Note that this ALTQ tag is only meaningful for packets going "out" of IPFW, and not being rejected or going to divert sockets. Note that if there is insufficient memory at the time the packet is processed, it will not be tagged, so it is wise to make your ALTQ "default" queue policy account for this. If multiple **altq** rules match a single packet, only the first one adds the ALTQ classification tag. In doing so, traffic may be shaped by using **count altq queue** rules for classification early in the ruleset, then later applying the filtering decision. For example, **check-state** and **keep-state** rules may come later and provide the actual filtering decisions in addition to the fallback ALTQ tag.

You must run `pfctl`(8) to set up the queues before IPFW will be able to look them up by name, and if the ALTQ disciplines are rearranged, the rules containing the queue identifiers in the kernel will likely have gone stale and need to be reloaded. Stale queue identifiers will probably result in misclassification.

All system ALTQ processing can be turned on or off via **ipfw enable altq** and **ipfw disable altq**. The usage of `net.inet.ipfw.one_pass` is irrelevant to ALTQ traffic shaping, as the actual rule action is followed always after adding an ALTQ tag.

RULE ACTIONS

A rule can be associated with one of the following actions, which will be executed when the packet matches the body of the rule.

allow | **accept** | **pass** | **permit**

Allow packets that match rule. The search terminates.

check-state

Checks the packet against the dynamic ruleset. If a match is found, execute the action associated with the rule which generated this dynamic rule, otherwise move to the next rule.

Check-state rules do not have a body. If no **check-state** rule is found, the dynamic ruleset is checked at the first **keep-state** or **limit** rule.

count Update counters for all packets that match rule. The search continues with the next rule.

deny | **drop**

Discard packets that match this rule. The search terminates.

divert *port*

Divert packets that match this rule to the `divert(4)` socket bound to port *port*. The search terminates.

 fwd | **forward** *ipaddr*[,*port*]

Change the next-hop on matching packets to *ipaddr*, which can be an IP address or a host name. The search terminates if this rule matches.

If *ipaddr* is a local address, then matching packets will be forwarded to *port* (or the port number in the packet if one is not specified in the rule) on the local machine.

If *ipaddr* is not a local address, then the port number (if specified) is ignored, and the packet will be forwarded to the remote address, using the route as found in the local routing table for that IP.

A *fwd* rule will not match layer-2 packets (those received on `ether_input`, `ether_output`, or bridged).

The **fwd** action does not change the contents of the packet at all. In particular, the destination address remains unmodified, so packets forwarded to another system will usually be rejected by that system unless there is a matching rule on that system to capture them. For packets forwarded locally, the local address of the socket will be set to the original destination address of the packet. This makes the `netstat(1)` entry look rather weird but is intended for use with transparent proxy servers.

To enable **fwd** a custom kernel needs to be compiled with the option **options IPFWALL_FORWARD**. With the additional option **options IPFWALL_FORWARD_EXTENDED** all safeguards are removed and it also makes it possible to redirect packets destined to locally configured IP addresses. Please note that such rules apply to locally generated packets as well and great care is required to ensure proper behaviour for automatically generated packets like ICMP message size exceeded and others.

pipe *pipe_nr*

Pass packet to a `dummynet(4)` “pipe” (for bandwidth limitation, delay, etc.). See the **TRAFFIC SHAPER (DUMMYPNET) CONFIGURATION** Section for further information. The search terminates; however, on exit from the pipe and if the `sysctl(8)` variable `net.inet.ip.fw.one_pass` is not set, the packet is passed again to the firewall code starting from the next rule.

queue *queue_nr*

Pass packet to a `dummynet(4)` “queue” (for bandwidth limitation using WF2Q+).

reject (Deprecated). Synonym for **unreach host**.

reset Discard packets that match this rule, and if the packet is a TCP packet, try to send a TCP reset (RST) notice. The search terminates.

reset6 Discard packets that match this rule, and if the packet is a TCP packet, try to send a TCP reset (RST) notice. The search terminates.

skipto *number*

Skip all subsequent rules numbered less than *number*. The search continues with the first rule numbered *number* or higher.

tee *port*

Send a copy of packets matching this rule to the `divert(4)` socket bound to port *port*. The search continues with the next rule.

unreach *code*

Discard packets that match this rule, and try to send an ICMP unreachable notice with code *code*, where *code* is a number from 0 to 255, or one of these aliases: **net**, **host**, **protocol**, **port**, **needfrag**, **srcfail**, **net-unknown**, **host-unknown**, **isolated**, **net-prohib**, **host-prohib**, **tosnet**, **toshost**, **filter-prohib**, **host-precedence** or **precedence-cutoff**. The search terminates.

unreach6 *code*

Discard packets that match this rule, and try to send an ICMPv6 unreachable notice with code *code*, where *code* is a number from 0, 1, 3 or 4, or one of these aliases: **no-route**, **admin-prohib**, **address** or **port**. The search terminates.

netgraph *cookie*

Divert packet into netgraph with given *cookie*. The search terminates. If packet is later returned from netgraph it is either accepted or continues with the next rule, depending on `net.inet.ip.fw.one_pass` sysctl variable.

ngtee *cookie*

A copy of packet is diverted into netgraph, original packet is either accepted or continues with the next rule, depending on `net.inet.ip.fw.one_pass` sysctl variable. See `ng_ipfw(4)` for more information on **netgraph** and **ngtee** actions.

RULE BODY

The body of a rule contains zero or more patterns (such as specific source and destination addresses or ports, protocol options, incoming or outgoing interfaces, etc.) that the packet must match in order to be recognised. In general, the patterns are connected by (implicit) **and** operators -- i.e., all must match in order for the rule to match. Individual patterns can be prefixed by the **not** operator to reverse the result of the match, as in

```
ipfw add 100 allow ip from not 1.2.3.4 to any
```

Additionally, sets of alternative match patterns (*or-blocks*) can be constructed by putting the patterns in lists enclosed between parentheses () or braces { }, and using the **or** operator as follows:

```
ipfw add 100 allow ip from { x or not y or z } to any
```

Only one level of parentheses is allowed. Beware that most shells have special meanings for parentheses or braces, so it is advisable to put a backslash \ in front of them to prevent such interpretations.

The body of a rule must in general include a source and destination address specifier. The keyword *any* can be used in various places to specify that the content of a required field is irrelevant.

The rule body has the following format:

```
[proto from src to dst] [options]
```

The first part (*proto from src to dst*) is for backward compatibility with earlier versions of FreeBSD. In modern FreeBSD any match pattern (including MAC headers, IP protocols, addresses and ports) can be specified in the *options* section.

Rule fields have the following meaning:

proto: *protocol* | { *protocol* **or** . . . }

protocol: [**not**]*protocol-name* | *protocol-number*

An IP protocol specified by number or name (for a complete list see */etc/protocols*), or one of the following keywords:

ip4 | **ipv4**
Matches IPv4 packets.

ip6 | **ipv6**
Matches IPv6 packets.

ip | **all**
Matches any packet.

The **ipv6** in **proto** option will be treated as inner protocol. And, the **ipv4** is not available in **proto** option.

The { *protocol* **or** . . . } format (an *or-block*) is provided for convenience only but its use is deprecated.

src and *dst*: {**addr** | { *addr* **or** . . . }} [[**not**]*ports*]

An address (or a list, see below) optionally followed by *ports* specifiers.

The second format (*or-block* with multiple addresses) is provided for convenience only and its use is discouraged.

addr: [**not**] {**any** | **me** | **me6** **table**(*number*[,*value*]) | *addr-list* | *addr-set*}

any matches any IP address.

me matches any IP address configured on an interface in the system.

me6 matches any IPv6 address configured on an interface in the system. The address list is evaluated at the time the packet is analysed.

table(*number*[,*value*])

Matches any IPv4 address for which an entry exists in the lookup table *number*. If an optional 32-bit unsigned *value* is also specified, an entry will match only if it has this value. See the **LOOKUP TABLES** section below for more information on lookup tables.

addr-list: *ip-addr*[,*addr-list*]

ip-addr:

A host or subnet address specified in one of the following ways:

numeric-ip | *hostname*

Matches a single IPv4 address, specified as dotted-quad or a hostname. Hostnames are resolved at the time the rule is added to the firewall list.

addr/masklen

Matches all addresses with base *addr* (specified as an IP address or a hostname) and mask width of **masklen** bits. As an example, 1.2.3.4/25 will match all IP numbers from 1.2.3.0 to 1.2.3.127.

addr:mask

Matches all addresses with base *addr* (specified as an IP address or a hostname) and the mask of *mask*, specified as a dotted quad. As an example, 1.2.3.4:255.0.255.0 will match 1.*.3.*. This form is advised only for non-contiguous masks. It is better to resort

to the *addr/masklen* format for contiguous masks, which is more compact and less error-prone.

addr-set: *addr*[/*masklen*]{*list*}

list: {*num* | *num-num*}[,*list*]

Matches all addresses with base address *addr* (specified as an IP address or a hostname) and whose last byte is in the list between braces { }. Note that there must be no spaces between braces and numbers (spaces after commas are allowed). Elements of the list can be specified as single entries or ranges. The *masklen* field is used to limit the size of the set of addresses, and can have any value between 24 and 32. If not specified, it will be assumed as 24.

This format is particularly useful to handle sparse address sets within a single rule. Because the matching occurs using a bitmask, it takes constant time and dramatically reduces the complexity of rulesets.

As an example, an address specified as 1.2.3.4/24{128,35-55,89} will match the following IP addresses:

1.2.3.128, 1.2.3.35 to 1.2.3.55, 1.2.3.89 .

addr6-list: *ip6-addr*[,*addr6-list*]

ip6-addr:

A host or subnet specified one of the following ways:

numeric-ip | *hostname*

Matches a single IPv6 address as allowed by `inet_pton(3)` or a hostname. Hostnames are resolved at the time the rule is added to the firewall list.

addr/masklen

Matches all IPv6 addresses with base *addr* (specified as allowed by `inet_pton` or a hostname) and mask width of **masklen** bits.

No support for sets of IPv6 addresses is provided because IPv6 addresses are typically random past the initial prefix.

ports: {*port* | *port-port*}[,*ports*]

For protocols which support port numbers (such as TCP and UDP), optional **ports** may be specified as one or more ports or port ranges, separated by commas but no spaces, and an optional **not** operator. The '-' notation specifies a range of ports (including boundaries).

Service names (from `/etc/services`) may be used instead of numeric port values. The length of the port list is limited to 30 ports or ranges, though one can specify larger ranges by using an *or-block* in the **options** section of the rule.

A backslash ('\') can be used to escape the dash ('-') character in a service name (from a shell, the backslash must be typed twice to avoid the shell itself interpreting it as an escape character).

```
ipfw add count tcp from any ftp\\\-data-ftp to any
```

Fragmented packets which have a non-zero offset (i.e., not the first fragment) will never match a rule which has one or more port specifications. See the **frag** option for details on matching fragmented packets.

RULE OPTIONS (MATCH PATTERNS)

Additional match patterns can be used within rules. Zero or more of these so-called *options* can be present in a rule, optionally prefixed by the **not** operand, and possibly grouped into *or-blocks*.

The following match patterns can be used (listed in alphabetical order):

// this is a comment.

Inserts the specified text as a comment in the rule. Everything following // is considered as a comment and stored in the rule. You can have comment-only rules, which are listed as having a **count** action followed by the comment.

bridged

Alias for **layer2**.

diverted

Matches only packets generated by a divert socket.

diverted-loopback

Matches only packets coming from a divert socket back into the IP stack input for delivery.

diverted-output

Matches only packets going from a divert socket back outward to the IP stack output for delivery.

dst-ip *ip-address*

Matches IPv4 packets whose destination IP is one of the address(es) specified as argument.

{dst-ip6 | dst-ipv6} *ip6-address*

Matches IPv6 packets whose destination IP is one of the address(es) specified as argument.

dst-port *ports*

Matches IP packets whose destination port is one of the port(s) specified as argument.

established

Matches TCP packets that have the RST or ACK bits set.

ext6hdr *header*

Matches IPv6 packets containing the extended header given by *header*. Supported headers are:

Fragment, (**frag**), Hop-to-hop options (**hopopt**), Source routing (**route**), Destination options (**dstopt**), IPsec authentication headers (**ah**), and IPsec encapsulated security payload headers (**esp**).

flow-id *labels*

Matches IPv6 packets containing any of the flow labels given in *labels*. *labels* is a comma separate list of numeric flow labels.

frag

Matches packets that are fragments and not the first fragment of an IP datagram. Note that these packets will not have the next protocol header (e.g. TCP, UDP) so options that look into these headers cannot match.

gid *group*

Matches all TCP or UDP packets sent by or received for a *group*. A *group* may be specified by name or number. This option should be used only if `debug.mpsafenet=0` to avoid possible deadlocks due to layering violations in its implementation.

jail *prisonID*

Matches all TCP or UDP packets sent by or received for the jail whose prison ID is *prisonID*. This option should be used only if `debug.mpsafenet=0` to avoid possible deadlocks due to layering violations in its implementation.

icmptypes *types*

Matches ICMP packets whose ICMP type is in the list *types*. The list may be specified as any combination of individual types (numeric) separated by commas. *Ranges are not allowed*. The supported ICMP types are:

echo reply (0), destination unreachable (3), source quench (4), redirect (5), echo request (8), router advertisement (9), router solicitation (10), time-to-live exceeded (11), IP header bad (12), timestamp request (13), timestamp reply (14), information request (15), information reply (16), address mask request (17) and address mask reply (18).

icmp6types *types*

Matches ICMP6 packets whose ICMP6 type is in the list of *types*. The list may be specified as any combination of individual types (numeric) separated by commas. *Ranges are not allowed.*

in | out

Matches incoming or outgoing packets, respectively. **in** and **out** are mutually exclusive (in fact, **out** is implemented as **not in**).

ipid *id-list*

Matches IPv4 packets whose **ip_id** field has value included in *id-list*, which is either a single value or a list of values or ranges specified in the same way as *ports*.

iplen *len-list*

Matches IP packets whose total length, including header and data, is in the set *len-list*, which is either a single value or a list of values or ranges specified in the same way as *ports*.

ipoptions *spec*

Matches packets whose IPv4 header contains the comma separated list of options specified in *spec*. The supported IP options are:

ssrr (strict source route), **lsrr** (loose source route), **rr** (record packet route) and **ts** (timestamp). The absence of a particular option may be denoted with a '!'.

ipprecedence *precedence*

Matches IPv4 packets whose precedence field is equal to *precedence*.

ipsec Matches packets that have IPSEC history associated with them (i.e., the packet comes encapsulated in IPSEC, the kernel has IPSEC support and IPSEC_FILTERGIF option, and can correctly decapsulate it).

Note that specifying **ipsec** is different from specifying **proto ipsec** as the latter will only look at the specific IP protocol field, irrespective of IPSEC kernel support and the validity of the IPSEC data.

Further note that this flag is silently ignored in kernels without IPSEC support. It does not affect rule processing when given and the rules are handled as if with no **ipsec** flag.

iptos *spec*

Matches IPv4 packets whose **tos** field contains the comma separated list of service types specified in *spec*. The supported IP types of service are:

lowdelay (IPTOS_LOWDELAY), **throughput** (IPTOS_THROUGHPUT), **reliability** (IPTOS_RELIABILITY), **mincost** (IPTOS_MINCOST), **congestion** (IPTOS_CE). The absence of a particular type may be denoted with a '!'.

ipttl *t11-list*

Matches IPv4 packets whose time to live is included in *t11-list*, which is either a single value or a list of values or ranges specified in the same way as *ports*.

ipversion *ver*

Matches IP packets whose IP version field is *ver*.

keep-state

Upon a match, the `fi` rewall will create a dynamic rule, whose default behaviour is to match bidirectional traffic between source and destination IP/port using the same protocol. The rule has a limited lifetime (controlled by a set of `sysctl(8)` variables), and the lifetime is refreshed every time a matching packet is found.

layer2 Matches only layer2 packets, i.e., those passed to `ipfw` from `ether_demux()` and `ether_output_frame()`.

limit {**src-addr** | **src-port** | **dst-addr** | **dst-port**} *N*

The `fi` rewall will only allow *N* connections with the same set of parameters as specified in the rule. One or more of source and destination addresses and ports can be specified. Currently, only IPv4 fw's are supported.

{ **MAC** | **mac** } *dst-mac src-mac*

Match packets with a given *dst-mac* and *src-mac* addresses, specified as the **any** keyword (matching any MAC address), or six groups of hex digits separated by colons, and optionally followed by a mask indicating the significant bits. The mask may be specified using either of the following methods:

1. A slash (/) followed by the number of significant bits. For example, an address with 33 significant bits could be specified as:

```
MAC 10:20:30:40:50:60/33 any
```

2. An ampersand (&) followed by a bitmask specified as six groups of hex digits separated by colons. For example, an address in which the last 16 bits are significant could be specified as:

```
MAC 10:20:30:40:50:60&00:00:00:00:ff:ff any
```

Note that the ampersand character has a special meaning in many shells and should generally be escaped.

Note that the order of MAC addresses (destination first, source second) is the same as on the wire, but the opposite of the one used for IP addresses.

mac-type *mac-type*

Matches packets whose Ethernet Type field corresponds to one of those specified as argument. *mac-type* is specified in the same way as **port numbers** (i.e., one or more comma-separated single values or ranges). You can use symbolic names for known values such as *vlan*, *ipv4*, *ipv6*. Values can be entered as decimal or hexadecimal (if prefixed by 0x), and they are always printed as hexadecimal (unless the **-N** option is used, in which case symbolic resolution will be attempted).

proto *protocol*

Matches packets with the corresponding IP protocol.

recv | **xmit** | **via** {*ifX* | *if** | *ipno* | *any*}

Matches packets received, transmitted or going through, respectively, the interface specified by exact name (*ifX*), by device name (*if**), by IP address, or through some interface.

The **via** keyword causes the interface to always be checked. If **recv** or **xmit** is used instead of **via**, then only the receive or transmit interface (respectively) is checked. By specifying both, it is possible to match packets based on both receive and transmit interface, e.g.:

```
ipfw add deny ip from any to any out recv ed0 xmit ed1
```

The **recv** interface can be tested on either incoming or outgoing packets, while the **xmit** interface can only be tested on outgoing packets. So **out** is required (and **in** is invalid) whenever

xmit is used.

A packet may not have a receive or transmit interface: packets originating from the local host have no receive interface, while packets destined for the local host have no transmit interface.

setup Matches TCP packets that have the SYN bit set but no ACK bit. This is the short form of “`tcpflags syn,!ack`”.

src-ip *ip-address*

Matches IPv4 packets whose source IP is one of the address(es) specified as an argument.

src-ip6 *ip6-address*

Matches IPv6 packets whose source IP is one of the address(es) specified as an argument.

src-port *ports*

Matches IP packets whose source port is one of the port(s) specified as argument.

tcpack *ack*

TCP packets only. Match if the TCP header acknowledgment number field is set to *ack*.

tcpdatalen *tcpdatalen-list*

Matches TCP packets whose length of TCP data is *tcpdatalen-list*, which is either a single value or a list of values or ranges specified in the same way as *ports*.

tcpflags *spec*

TCP packets only. Match if the TCP header contains the comma separated list of flags specified in *spec*. The supported TCP flags are:

fin, **syn**, **rst**, **psh**, **ack** and **urg**. The absence of a particular flag may be denoted with a ‘!’. A rule which contains a **tcpflags** specification can never match a fragmented packet which has a non-zero offset. See the **frag** option for details on matching fragmented packets.

tcpseq *seq*

TCP packets only. Match if the TCP header sequence number field is set to *seq*.

tcpwin *win*

TCP packets only. Match if the TCP header window field is set to *win*.

tcpoptions *spec*

TCP packets only. Match if the TCP header contains the comma separated list of options specified in *spec*. The supported TCP options are:

mss (maximum segment size), **window** (tcp window advertisement), **sack** (selective ack), **ts** (rfc1323 timestamp) and **cc** (rfc1644 t/tcp connection count). The absence of a particular option may be denoted with a ‘!’.

uid *user*

Match all TCP or UDP packets sent by or received for a *user*. A *user* may be matched by name or identification number. This option should be used only if `debug.mpsafenet=0` to avoid possible deadlocks due to layering violations in its implementation.

verrevpath

For incoming packets, a routing table lookup is done on the packet's source address. If the interface on which the packet entered the system matches the outgoing interface for the route, the packet matches. If the interfaces do not match up, the packet does not match. All outgoing packets or packets with no incoming interface match.

The name and functionality of the option is intentionally similar to the Cisco IOS command:

```
ip verify unicast reverse-path
```

This option can be used to make anti-spoofing rules to reject all packets with source addresses not from this interface. See also the option **antispoof**.

verssreach

For incoming packets, a routing table lookup is done on the packet's source address. If a route to the source address exists, but not the default route or a blackhole/reject route, the packet matches. Otherwise, the packet does not match. All outgoing packets match.

The name and functionality of the option is intentionally similar to the Cisco IOS command:

```
ip verify unicast source reachable-via any
```

This option can be used to make anti-spoofing rules to reject all packets whose source address is unreachable.

antispoof

For incoming packets, the packet's source address is checked if it belongs to a directly connected network. If the network is directly connected, then the interface the packet came on in is compared to the interface the network is connected to. When incoming interface and directly connected interface are not the same, the packet does not match. Otherwise, the packet does match. All outgoing packets match.

This option can be used to make anti-spoofing rules to reject all packets that pretend to be from a directly connected network but do not come in through that interface. This option is similar to but more restricted than **verrevpath** because it engages only on packets with source addresses of directly connected networks instead of all source addresses.

LOOKUP TABLES

Lookup tables are useful to handle large sparse address sets, typically from a hundred to several thousands of entries. There may be up to 128 different lookup tables, numbered 0 to 127.

Each entry is represented by an *addr[/masklen]* and will match all addresses with base *addr* (specified as an IP address or a hostname) and mask width of *masklen* bits. If *masklen* is not specified, it defaults to 32. When looking up an IP address in a table, the most specific entry will match. Associated with each entry is a 32-bit unsigned *value*, which can optionally be checked by a rule matching code. When adding an entry, if *value* is not specified, it defaults to 0.

An entry can be added to a table (**add**), removed from a table (**delete**), a table can be examined (**list**) or flushed (**flush**).

Internally, each table is stored in a Radix tree, the same way as the routing table (see `route(4)`).

Lookup tables currently support IPv4 addresses only.

The **tablearg** feature provides the ability to use a value, looked up in the table, as the argument for a rule action. This can significantly reduce number of rules in some configurations. The **tablearg** argument can be used with the following actions: **pipe**, **queue**, **divert**, **tee**, **netgraph**, **ngtee**. See the **EXAMPLES** Section for example usage of tables and the **tablearg** keyword.

SETS OF RULES

Each rule belongs to one of 32 different *sets*, numbered 0 to 31. Set 31 is reserved for the default rule.

By default, rules are put in set 0, unless you use the **set N** attribute when entering a new rule. Sets can be individually and atomically enabled or disabled, so this mechanism permits an easy way to store multiple configurations of the firewall and quickly (and atomically) switch between them. The command to enable/disable sets is

```
ipfw set [disable number . . .] [enable number . . .]
```

where multiple **enable** or **disable** sections can be specified. Command execution is atomic on all the sets specified in the command. By default, all sets are enabled.

When you disable a set, its rules behave as if they do not exist in the firewall configuration, with only one exception:

dynamic rules created from a rule before it had been disabled will still be active until they expire. In order to delete dynamic rules you have to explicitly delete the parent rule which generated them.

The set number of rules can be changed with the command

```
ipfw set move {rule rule-number | old-set} to new-set
```

Also, you can atomically swap two rulesets with the command

```
ipfw set swap first-set second-set
```

See the **EXAMPLES** Section on some possible uses of sets of rules.

STATEFUL FIREWALL

Stateful operation is a way for the firewall to dynamically create rules for specific flows when packets that match a given pattern are detected. Support for stateful operation comes through the **check-state**, **keep-state** and **limit** options of **rules**.

Dynamic rules are created when a packet matches a **keep-state** or **limit** rule, causing the creation of a *dynamic* rule which will match all and only packets with a given *protocol* between a *src-ip/src-port* *dst-ip/dst-port* pair of addresses (*src* and *dst* are used here only to denote the initial match addresses, but they are completely equivalent afterwards). Dynamic rules will be checked at the first **check-state**, **keep-state** or **limit** occurrence, and the action performed upon a match will be the same as in the parent rule.

Note that no additional attributes other than protocol and IP addresses and ports are checked on dynamic rules.

The typical use of dynamic rules is to keep a closed firewall configuration, but let the first TCP SYN packet from the inside network install a dynamic rule for the flow so that packets belonging to that session will be allowed through the firewall:

```
ipfw add check-state
ipfw add allow tcp from my-subnet to any setup keep-state
ipfw add deny tcp from any to any
```

A similar approach can be used for UDP, where an UDP packet coming from the inside will install a dynamic rule to let the response through the firewall:

```
ipfw add check-state
ipfw add allow udp from my-subnet to any keep-state
ipfw add deny udp from any to any
```

Dynamic rules expire after some time, which depends on the status of the flow and the setting of some **sysctl** variables. See Section **SYSCTL VARIABLES** for more details. For TCP sessions, dynamic rules can be instructed to periodically send keepalive packets to refresh the state of the rule when it is about to expire.

See Section **EXAMPLES** for more examples on how to use dynamic rules.

TRAFFIC SHAPER (DUMMYNET) CONFIGURATION

ipfw is also the user interface for the `dummynet(4)` traffic shaper.

dummynet operates by first using the firewall to classify packets and divide them into *flows*, using any match pattern that can be used in **ipfw** rules. Depending on local policies, a flow can contain packets for a single TCP connection, or from/to a given host, or entire subnet, or a protocol type, etc.

Packets belonging to the same flow are then passed to either of two different objects, which implement the traffic regulation:

- pipe* A pipe emulates a link with given bandwidth, propagation delay, queue size and packet loss rate. Packets are queued in front of the pipe as they come out from the classifier, and then transferred to the pipe according to the pipe's parameters.
- queue* A queue is an abstraction used to implement the WF2Q+ (Worst-case Fair Weighted Fair Queueing) policy, which is an efficient variant of the WFQ policy. The queue associates a *weight* and a reference pipe to each flow, and then all backlogged (i.e., with packets queued) flows linked to the same pipe share the pipe's bandwidth proportionally to their weights. Note that weights are not priorities; a flow with a lower weight is still guaranteed to get its fraction of the bandwidth even if a flow with a higher weight is permanently backlogged.

In practice, *pipes* can be used to set hard limits to the bandwidth that a flow can use, whereas *queues* can be used to determine how different flows share the available bandwidth.

The *pipe* and *queue* configuration commands are the following:

```
pipe number config pipe-configuration
queue number config queue-configuration
```

The following parameters can be configured for a pipe:

bw *bandwidth* | *device*

Bandwidth, measured in [K|M]{bit/s|Byte/s}.

A value of 0 (default) means unlimited bandwidth. The unit must immediately follow the number, as in

```
ipfw pipe 1 config bw 300Kbit/s
```

If a device name is specified instead of a numeric value, as in

```
ipfw pipe 1 config bw tun0
```

then the transmit clock is supplied by the specified device. At the moment only the `tun(4)` device supports this functionality, for use in conjunction with `ppp(8)`.

delay *ms-delay*

Propagation delay, measured in milliseconds. The value is rounded to the next multiple of the clock tick (typically 10ms, but it is a good practice to run kernels with "options HZ=1000" to reduce the granularity to 1ms or less). Default value is 0, meaning no delay.

The following parameters can be configured for a queue:

pipe *pipe_nr*

Connects a queue to the specified pipe. Multiple queues (with the same or different weights) can be connected to the same pipe, which specifies the aggregate rate for the set of queues.

weight *weight*

Specifies the weight to be used for fbws matching this queue. The weight must be in the range 1..100, and defaults to 1.

Finally, the following parameters can be configured for both pipes and queues:

buckets *hash-table-size*

Specifies the size of the hash table used for storing the various queues. Default value is 64 controlled by the `sysctl(8)` variable `net.inet.ip.dummynet.hash_size`, allowed range is 16 to 65536.

mask *mask-specifier*

Packets sent to a given pipe or queue by an **ipfw** rule can be further classified into multiple fbws, each of which is then sent to a different *dynamic* pipe or queue. A fbw identifier is constructed by masking the IP addresses, ports and protocol types as specified with the **mask** options in the configuration of the pipe or queue. For each different fbw identifier, a new pipe or queue is created with the same parameters as the original object, and matching packets are sent to it.

Thus, when *dynamic pipes* are used, each fbw will get the same bandwidth as defined by the pipe, whereas when *dynamic queues* are used, each fbw will share the parent's pipe bandwidth evenly with other fbws generated by the same queue (note that other queues with different weights might be connected to the same pipe).

Available mask specifiers are a combination of one or more of the following:

dst-ip *mask*, **dst-ip6** *mask*, **src-ip** *mask*, **src-ip6** *mask*, **dst-port** *mask*, **src-port** *mask*, **flow-id** *mask*, **proto** *mask* or **all**,

where the latter means all bits in all fields are significant.

noerror

When a packet is dropped by a dummynet queue or pipe, the error is normally reported to the caller routine in the kernel, in the same way as it happens when a device queue fills up. Setting this option reports the packet as successfully delivered, which can be needed for some experimental setups where you want to simulate loss or congestion at a remote router.

plr *packet-loss-rate*

Packet loss rate. Argument *packet-loss-rate* is a floating-point number between 0 and 1, with 0 meaning no loss, 1 meaning 100% loss. The loss rate is internally represented on 31 bits.

queue {*slots* | *sizeKbytes*}

Queue size, in *slots* or **KBytes**. Default value is 50 slots, which is the typical queue size for Ethernet devices. Note that for slow speed links you should keep the queue size short or your traffic might be affected by a significant queueing delay. E.g., 50 max-sized ethernet packets (1500 bytes) mean 600Kbit or 20s of queue on a 30Kbit/s pipe. Even worse effects can result if you get packets from an interface with a much larger MTU, e.g. the loopback interface with its 16KB packets.

red | **gred** *w_q/min_th/max_th/max_p*

Make use of the RED (Random Early Detection) queue management algorithm. *w_q* and *max_p* are floating point numbers between 0 and 1 (0 not included), while *min_th* and *max_th* are integer numbers specifying thresholds for queue management (thresholds are computed in bytes if the queue has been defined in bytes, in slots otherwise). The dummynet(4) also supports the gentle RED variant (gred). Three `sysctl(8)` variables can be used to control the RED behaviour:

net.inet.ip.dummynet.red_lookup_depth

specifies the accuracy in computing the average queue when the link is idle (defaults to 256, must be greater than zero)

net.inet.ip.dummynet.red_avg_pkt_size

specifies the expected average packet size (defaults to 512, must be greater than zero)

net.inet.ip.dummynet.red_max_pkt_size

specifies the expected maximum packet size, only used when queue thresholds are in bytes (defaults to 1500, must be greater than zero).

When used with IPv6 data, dummynet currently has several limitations. First, `debug.mpsafenet=0` must be set. Second, the information necessary to route link-local packets to an interface is not available after processing by dummynet so those packets are dropped in the output path. Care should be taken to insure that link-local packets are not passed to dummynet.

CHECKLIST

Here are some important points to consider when designing your rules:

- Remember that you filter both packets going **in** and **out**. Most connections need packets going in both directions.
- Remember to test very carefully. It is a good idea to be near the console when doing this. If you cannot be near the console, use an auto-recovery script such as the one in `/usr/share/examples/ipfw/change_rules.sh`.
- Do not forget the loopback interface.

FINE POINTS

- There are circumstances where fragmented datagrams are unconditionally dropped. TCP packets are dropped if they do not contain at least 20 bytes of TCP header, UDP packets are dropped if they do not contain a full 8 byte UDP header, and ICMP packets are dropped if they do not contain 4 bytes of ICMP header, enough to specify the ICMP type, code, and checksum. These packets are simply logged as “pullup failed” since there may not be enough good data in the packet to produce a meaningful log entry.
- Another type of packet is unconditionally dropped, a TCP packet with a fragment offset of one. This is a valid packet, but it only has one use, to try to circumvent firewalls. When logging is enabled, these packets are reported as being dropped by rule -1.
- If you are logged in over a network, loading the `kld(4)` version of **ipfw** is probably not as straightforward as you would think. I recommend the following command line:

```
kldload ipfw && \
ipfw add 32000 allow ip from any to any
```

Along the same lines, doing an

```
ipfw flush
```

in similar surroundings is also a bad idea.

- The **ipfw** filter list may not be modified if the system security level is set to 3 or higher (see `init(8)` for information on system security levels).

PACKET DIVERSION

A `divert(4)` socket bound to the specified port will receive all packets diverted to that port. If no socket is bound to the destination port, or if the divert module is not loaded, or if the kernel was not compiled with divert socket support, the packets are dropped.

SYSCTL VARIABLES

A set of `sysctl(8)` variables controls the behaviour of the firewall and associated modules (**dummynet**, **bridge**). These are shown below together with their default value (but always check with the `sysctl(8)` command what value is actually in use) and meaning:

net.inet.ip.dummynet.expire: 1

Lazily delete dynamic pipes/queue once they have no pending traffic. You can disable this by setting the variable to 0, in which case the pipes/queues will only be deleted when the threshold is reached.

net.inet.ip.dummynet.hash_size: 64

Default size of the hash table used for dynamic pipes/queues. This value is used when no **buckets** option is specified when configuring a pipe/queue.

net.inet.ip.dummynet.max_chain_len: 16

Target value for the maximum number of pipes/queues in a hash bucket. The product **max_chain_len*hash_size** is used to determine the threshold over which empty pipes/queues will be expired even when **net.inet.ip.dummynet.expire=0**.

net.inet.ip.dummynet.red_lookup_depth: 256

net.inet.ip.dummynet.red_avg_pkt_size: 512

net.inet.ip.dummynet.red_max_pkt_size: 1500

Parameters used in the computations of the drop probability for the RED algorithm.

net.inet.ip.fw.autoinc_step: 100

Delta between rule numbers when auto-generating them. The value must be in the range 1..1000.

net.inet.ip.fw.curr_dyn_buckets: *net.inet.ip.fw.dyn_buckets*

The current number of buckets in the hash table for dynamic rules (readonly).

net.inet.ip.fw.debug: 1

Controls debugging messages produced by **ipfw**.

net.inet.ip.fw.dyn_buckets: 256

The number of buckets in the hash table for dynamic rules. Must be a power of 2, up to 65536. It only takes effect when all dynamic rules have expired, so you are advised to use a **flush** command to make sure that the hash table is resized.

net.inet.ip.fw.dyn_count: 3

Current number of dynamic rules (read-only).

net.inet.ip.fw.dyn_keepalive: 1

Enables generation of keepalive packets for **keep-state** rules on TCP sessions. A keepalive is generated to both sides of the connection every 5 seconds for the last 20 seconds of the lifetime of the rule.

net.inet.ip.fw.dyn_max: 8192

Maximum number of dynamic rules. When you hit this limit, no more dynamic rules can be installed until old ones expire.

net.inet.ip.fw.dyn_ack_lifetime: 300

net.inet.ip.fw.dyn_syn_lifetime: 20

net.inet.ip.fw.dyn_fin_lifetime: 1

net.inet.ip.fw.dyn_rst_lifetime: 1

net.inet.ip.fw.dyn_udp_lifetime: 5

net.inet.ip.fw.dyn_short_lifetime: 30

These variables control the lifetime, in seconds, of dynamic rules. Upon the initial SYN exchange the lifetime is kept short, then increased after both SYN have been seen, then decreased again during the final FIN exchange or when a RST is received. Both *dyn_fin_lifetime* and *dyn_rst_lifetime* must be strictly lower than 5 seconds, the period of repetition of keepalives. The firewall enforces that.

net.inet.ip.fw.enable: 1

Enables the firewall. Setting this variable to 0 lets you run your machine without firewall even if compiled in.

net.inet.ip.fw.one_pass: 1

When set, the packet exiting from the `dummynet(4)` pipe or from `ng_ipfw(4)` node is not passed through the firewall again. Otherwise, after an action, the packet is reinjected into the firewall at the next rule.

net.inet.ip.fw.verbose: 1

Enables verbose messages.

net.inet.ip.fw.verbose_limit: 0

Limits the number of messages produced by a verbose firewall.

net.inet6.ip6.fw.deny_unknown_exthdrs: 1

If enabled packets with unknown IPv6 Extension Headers will be denied.

net.link.ether.ipfw: 0

Controls whether layer-2 packets are passed to `ipfw`. Default is no.

net.link.ether.bridge_ipfw: 0

Controls whether bridged packets are passed to `ipfw`. Default is no.

EXAMPLES

There are far too many possible uses of `ipfw` so this Section will only give a small set of examples.

BASIC PACKET FILTERING

This command adds an entry which denies all tcp packets from `cracker.evil.org` to the telnet port of `wolf.tambov.su` from being forwarded by the host:

```
ipfw add deny tcp from cracker.evil.org to wolf.tambov.su telnet
```

This one disallows any connection from the entire cracker's network to my host:

```
ipfw add deny ip from 123.45.67.0/24 to my.host.org
```

A first and efficient way to limit access (not using dynamic rules) is the use of the following rules:

```
ipfw add allow tcp from any to any established
ipfw add allow tcp from net1 portlist1 to net2 portlist2 setup
ipfw add allow tcp from net3 portlist3 to net3 portlist3 setup
...
ipfw add deny tcp from any to any
```

The first rule will be a quick match for normal TCP packets, but it will not match the initial SYN packet, which will be matched by the `setup` rules only for selected source/destination pairs. All other SYN packets will be rejected by the final `deny` rule.

If you administer one or more subnets, you can take advantage of the address sets and or-blocks and write extremely compact rulesets which selectively enable services to blocks of clients, as below:

```
goodguys="{ 10.1.2.0/24{20,35,66,18} or 10.2.3.0/28{6,3,11} }"
badguys="10.1.2.0/24{8,38,60}"

ipfw add allow ip from ${goodguys} to any
ipfw add deny ip from ${badguys} to any
... normal policies ...
```

The **verrevpath** option could be used to do automated anti-spoofing by adding the following to the top of a ruleset:

```
ipfw add deny ip from any to any not verrevpath in
```

This rule drops all incoming packets that appear to be coming to the system on the wrong interface. For example, a packet with a source address belonging to a host on a protected internal network would be dropped if it tried to enter the system from an external interface.

The **antispoof** option could be used to do similar but more restricted anti-spoofing by adding the following to the top of a ruleset:

```
ipfw add deny ip from any to any not antispoof in
```

This rule drops all incoming packets that appear to be coming from another directly connected system but on the wrong interface. For example, a packet with a source address of 192.168.0.0/24, configured on fxp0, but coming in on fxp1 would be dropped.

DYNAMIC RULES

In order to protect a site from flood attacks involving fake TCP packets, it is safer to use dynamic rules:

```
ipfw add check-state
ipfw add deny tcp from any to any established
ipfw add allow tcp from my-net to any setup keep-state
```

This will let the firewall install dynamic rules only for those connections which start with a regular SYN packet coming from the inside of our network. Dynamic rules are checked when encountering the first **check-state** or **keep-state** rule. A **check-state** rule should usually be placed near the beginning of the ruleset to minimize the amount of work scanning the ruleset. Your mileage may vary.

To limit the number of connections a user can open you can use the following type of rules:

```
ipfw add allow tcp from my-net/24 to any setup limit src-addr 10
ipfw add allow tcp from any to me setup limit src-addr 4
```

The former (assuming it runs on a gateway) will allow each host on a /24 network to open at most 10 TCP connections. The latter can be placed on a server to make sure that a single client does not use more than 4 simultaneous connections.

BEWARE: stateful rules can be subject to denial-of-service attacks by a SYN-flood which opens a huge number of dynamic rules. The effects of such attacks can be partially limited by acting on a set of `sysctl(8)` variables which control the operation of the firewall.

Here is a good usage of the **list** command to see accounting records and timestamp information:

```
ipfw -at list
```

or in short form without timestamps:

```
ipfw -a list
```

which is equivalent to:

```
ipfw show
```

Next rule diverts all incoming packets from 192.168.2.0/24 to divert port 5000:

```
ipfw divert 5000 ip from 192.168.2.0/24 to any in
```

TRAFFIC SHAPING

The following rules show some of the applications of **ipfw** and **dummynet(4)** for simulations and the like.

This rule drops random incoming packets with a probability of 5%:

```
ipfw add prob 0.05 deny ip from any to any in
```

A similar effect can be achieved making use of **dummynet** pipes:

```
ipfw add pipe 10 ip from any to any
ipfw pipe 10 config plr 0.05
```

We can use pipes to artificially limit bandwidth, e.g. on a machine acting as a router, if we want to limit traffic from local clients on 192.168.2.0/24 we do:

```
ipfw add pipe 1 ip from 192.168.2.0/24 to any out
ipfw pipe 1 config bw 300Kbit/s queue 50KBytes
```

note that we use the **out** modifier so that the rule is not used twice. Remember in fact that **ipfw** rules are checked both on incoming and outgoing packets.

Should we want to simulate a bidirectional link with bandwidth limitations, the correct way is the following:

```
ipfw add pipe 1 ip from any to any out
ipfw add pipe 2 ip from any to any in
ipfw pipe 1 config bw 64Kbit/s queue 10Kbytes
ipfw pipe 2 config bw 64Kbit/s queue 10Kbytes
```

The above can be very useful, e.g. if you want to see how your fancy Web page will look for a residential user who is connected only through a slow link. You should not use only one pipe for both directions, unless you want to simulate a half-duplex medium (e.g. AppleTalk, Ethernet, IRDA). It is not necessary that both pipes have the same configuration, so we can also simulate asymmetric links.

Should we want to verify network performance with the RED queue management algorithm:

```
ipfw add pipe 1 ip from any to any
ipfw pipe 1 config bw 500Kbit/s queue 100 red 0.002/30/80/0.1
```

Another typical application of the traffic shaper is to introduce some delay in the communication. This can significantly affect applications which do a lot of Remote Procedure Calls, and where the round-trip-time of the connection often becomes a limiting factor much more than bandwidth:

```
ipfw add pipe 1 ip from any to any out
ipfw add pipe 2 ip from any to any in
ipfw pipe 1 config delay 250ms bw 1Mbit/s
ipfw pipe 2 config delay 250ms bw 1Mbit/s
```

Per-flow queueing can be useful for a variety of purposes. A very simple one is counting traffic:

```
ipfw add pipe 1 tcp from any to any
ipfw add pipe 1 udp from any to any
ipfw add pipe 1 ip from any to any
```

```
ipfw pipe 1 config mask all
```

The above set of rules will create queues (and collect statistics) for all traffic. Because the pipes have no limitations, the only effect is collecting statistics. Note that we need 3 rules, not just the last one, because when **ipfw** tries to match IP packets it will not consider ports, so we would not see connections on separate ports as different ones.

A more sophisticated example is limiting the outbound traffic on a net with per-host limits, rather than per-network limits:

```
ipfw add pipe 1 ip from 192.168.2.0/24 to any out
ipfw add pipe 2 ip from any to 192.168.2.0/24 in
ipfw pipe 1 config mask src-ip 0x000000ff bw 200Kbit/s queue
20Kbytes
ipfw pipe 2 config mask dst-ip 0x000000ff bw 200Kbit/s queue
20Kbytes
```

LOOKUP TABLES

In the following example, we need to create several traffic bandwidth classes and we need different hosts/networks to fall into different classes. We create one pipe for each class and configure them accordingly. Then we create a single table and fill it with IP subnets and addresses. For each subnet/host we set the argument equal to the number of the pipe that it should use. Then we classify traffic using a single rule:

```
ipfw pipe 1 config bw 1000Kbyte/s
ipfw pipe 4 config bw 4000Kbyte/s
...
ipfw table 1 add 192.168.2.0/24 1
ipfw table 1 add 192.168.0.0/27 4
ipfw table 1 add 192.168.0.2 1
...
ipfw pipe tablearg ip from table(1) to any
```

SETS OF RULES

To add a set of rules atomically, e.g. set 18:

```
ipfw set disable 18
ipfw add NN set 18 ... # repeat as needed
ipfw set enable 18
```

To delete a set of rules atomically the command is simply:

```
ipfw delete set 18
```

To test a ruleset and disable it and regain control if something goes wrong:

```
ipfw set disable 18
ipfw add NN set 18 ... # repeat as needed
ipfw set enable 18; echo done; sleep 30 && ipfw set disable 18
```

Here if everything goes well, you press control-C before the "sleep" terminates, and your ruleset will be left active. Otherwise, e.g. if you cannot access your box, the ruleset will be disabled after the sleep terminates thus restoring the previous situation.

SEE ALSO

cpp(1), m4(1), altq(4), bridge(4), divert(4), dumynet(4), ip(4), ipfirewall(4), ng_ipfw(4), protocols(5), services(5), init(8), kldload(8), reboot(8), sysctl(8), syslogd(8)

HISTORY

The **ipfw** utility first appeared in FreeBSD 2.0. `dummynet(4)` was introduced in FreeBSD 2.2.8. Stateful extensions were introduced in FreeBSD 4.0. **ipfw2** was introduced in Summer 2002.

AUTHORS

Ugen J. S. Antsilevich,
Poul-Henning Kamp,
Alex Nash,
Archie Cobbs,
Luigi Rizzo.

API based upon code written by Daniel Boulet for BSDI.

Work on `dummynet(4)` traffic shaper supported by Akamba Corp.

BUGS

Use of `dummynet` with IPv6 requires that `debug.mpsafenet` be set to 0.

The syntax has grown over the years and sometimes it might be confusing. Unfortunately, backward compatibility prevents cleaning up mistakes made in the definition of the syntax.

!!! WARNING !!!

Misconfiguring the firewall can put your computer in an unusable state, possibly shutting down network services and requiring console access to regain control of it.

Incoming packet fragments diverted by **divert** are reassembled before delivery to the socket. The action used on those packet is the one from the rule which matches the first fragment of the packet.

Packets diverted to userland, and then reinserted by a userland process may lose various packet attributes. The packet source interface name will be preserved if it is shorter than 8 bytes and the userland process saves and reuses the `sockaddr_in` (as does `natd(8)`); otherwise, it may be lost. If a packet is reinserted in this manner, later rules may be incorrectly applied, making the order of **divert** rules in the rule sequence very important.

`Dummynet` drops all packets with IPv6 link-local addresses.

Rules using **uid** or **gid** may not behave as expected. In particular, incoming SYN packets may have no uid or gid associated with them since they do not yet belong to a TCP connection, and the uid/gid associated with a packet may not be as expected if the associated process calls `setuid(2)` or similar system calls.

Rules which use uid, gid or jail based matching should be used only if `debug.mpsafenet=0` to avoid possible deadlocks due to layering violations in its implementation.