/etc/pf.conf

Packet filter configuration file

Name:

`/etc/pf.conf`

Description:

The pf packet filter modifies, drops or passes packets according to rules or definitions specified inpf.conf. The default location of this file is /etc/pf.conf.

Statement order

The pf.conf file can include the following types of statements:

Macros
: User-defined variables may be defined and used later, simplifying the configuration file. Macros must be defined before they can be referenced in pf.conf.

Tables
: Tables provide a mechanism for increasing the performance and flexibility of rules with large numbers of source or destination addresses.

Options
: Options tune the behavior of the packet filtering engine.

Traffic Normalization (e.g. scrub)
: Traffic normalization protects internal machines against inconsistencies in Internet protocols and implementations.

Queueing
: Queueing provides rule-based bandwidth control.

Translation (Various forms of NAT)
: Translation rules specify how addresses are to be mapped or redirected to other addresses.

Packet Filtering
: Stateful and stateless packet filtering provides rule-based blocking or passing of packets.

With the exception of macros and tables, the types of statements should be grouped and appear inpf.conf in the order shown above, as this matches the operation of the underlying packet filtering engine. By default, pfctl enforces this order (see set require-order below).

Macros

Much as for cpp or m4, you can define macros that will later be expanded in context. Macro names must start with a letter, and may contain letters, digits and underscores. Macro names may not be reserved words (for example, pass, in, out). Macros aren't expanded inside quotes.
For example:

```
ext_if = "kue0"

all_ifs = "{" $ext_if lo0 "}"

pass out on $ext_if from any to any keep state

pass in  on $ext_if proto tcp from any to any port 25 keep state
```

Tables

Tables are named structures that can hold a collection of addresses and networks. Lookups against tables in pf are relatively fast, making a single rule with tables much more efficient, in terms of processor usage and memory consumption, than a large number of rules that differ only in IP address (either created explicitly or automatically by rule expansion).
You can use tables as the source or destination of filter rules, scrub rules or translation rules such asnat or rdr (see below for details on the various rule types) You can also use tables for the redirect address of nat and rdr rules and in the routing options of filter rules, but only for round-robin pools.
You can define tables with any of the following pfctl mechanisms. As with macros, reserved words may not be used as table

names.

Manually

Persistent tables can be manually created with the add or replace option of pfctl, before or after the rule set has been loaded.

Via pf.conf

Table definitions can be placed directly in this file, and loaded at the same time as other rules are loaded, atomically. Table definitions inside pf.conf use the table statement, and are especially useful to define non-persistent tables. The contents of a pre-existing table defined without a list of addresses to initialize it isn't altered when pf.conf is loaded. A table initialized with the empty list, { }, is cleared on loading.

Tables may be defined with the following attributes:

persist

Force io-pkt to keep the table even when no rules refer to it. If the flag isn't set, io-pkt automatically removes the table when the last rule referring to it is flushed.

const

Prevent the user from altering the contents of the table once it's been created. Without this flag, you can use pfctl to add or remove addresses from the table at any time, even when running withsecurelevel = 2.

For example:

```
table <private> const { 10/8, 172.16/12, 192.168/16 }

table <badhosts> persist

block on fxp0 from { <private>, <badhosts> } to any
```

creates a table called private, to hold RFC 1918 private network blocks, and a table called badhosts, which is initially empty. A filter rule is set up to block all traffic coming from addresses listed in either table.
The private table can't have its contents changed, and the badhosts table will exist even when no active filter rules refer to it. Addresses may later be added to the badhosts table, so that traffic from these hosts can be blocked by using:

```
# pfctl -t badhosts -Tadd 204.92.77.111
```

A table can also be initialized with an address list specified in one or more external files, using the following syntax:

```
table <spam> persist file "/etc/spammers" file "/etc/openrelays"

block on fxp0 from <spam> to any
```

The files /etc/spammers and /etc/openrelays list IP addresses, one per line. Any lines beginning with a # are treated as comments and ignored. In addition to being specified by IP address, hosts may also be specified by their hostname. When the resolver is called to add a hostname to a table, all resulting IPv4 and IPv6 addresses are placed into the table. IP addresses can also be entered in a table by specifying a valid interface name or the self keyword, in which case all addresses assigned to the interface(s) are added to the table.

Options

You can tune pf for various situations by using the set command:

set timeout *arguments*

Set the timeout specified by the arguments:

- interval — the interval between the purging of expired states and fragments.
- frag — the number of seconds before an unassembled fragment is expired.
- src.track — the length of time to retain a source-tracking entry after the last state expires.

When a packet matches a stateful connection, the seconds to live for the connection is updated to that of the proto.modifier that corresponds to the connection state. Each packet that matches this state resets the TTL. Tuning these values may improve the performance of the firewall, at the risk of dropping valid idle connections.

- tcp.first — the state after the first packet.
- tcp.opening — the state before the destination host ever sends a packet.
- tcp.established — the fully established state.
- tcp.closing — the state after the first FIN has been sent.
- tcp.finwait — the state after both FINs have been exchanged and the connection is closed. Some hosts (notably web servers on Solaris) send TCP packets even after closing the connection. Increasing the value of tcp.finwait (and possibly that of tcp.closing) can prevent blocking of such packets.
- tcp.closed — the state after one endpoint sends an RST.

ICMP and UDP are handled in a fashion similar to TCP, but with a much more limited set of states:

- udp.first — the state after the first packet.
- udp.single — the state if the source host sends more than one packet but the destination host has never sent one back.
- udp.multiple — the state if both hosts have sent packets.
- icmp.first — the state after the first packet.
- icmp.error — the state after an ICMP error came back in response to an ICMP packet.

Other protocols are handled similarly to UDP:

- other.first
- other.single
- other.multiple

Timeout values can be reduced adaptively as the number of state table entries grows.

- adaptive.start — when the number of state entries exceeds this value, adaptive scaling begins. All timeout values are scaled linearly with factor (adaptive.end − number of states) / (adaptive.end − adaptive.start).
- adaptive.end — when reaching this number of state entries, all timeout values become zero, effectively purging all state entries immediately. This value is used to define the scale factor; it shouldn't actually be reached (set a lower state limit, see below).

You can define these values both globally and for each rule. When used on a per-rule basis, the values relate to the number of states created by the rule, otherwise to the total number of states. For example:

```
set timeout tcp.first 120

set timeout tcp.established 86400

set timeout { adaptive.start 6000, adaptive.end 12000 }

set limit states 10000
```

With 9000 state table entries, the timeout values are scaled to 50% (tcp.first 60,tcp.established 43200).

set loginterface

Enable the collection of packet and byte count statistics for the given interface. To view these statistics, use:

```
pfctl -s info
```

```

```

In this example, pf collects statistics on the interface named dc0:

```
set loginterface dc0
```

You can disable the log interface by using:

```
set loginterface none
```

set limit

Set hard limits on the memory pools used by the packet filter. For example:

```
set limit states 20000
```

sets the maximum number of entries in the memory pool used by state table entries (generated by keep state rules) to 20000. This command:

```
set limit frags 20000
```

sets the maximum number of entries in the memory pool used for fragment reassembly (generated by scrub rules) to 20000. This command:

```
set limit src-nodes 2000
```

sets the maximum number of entries in the memory pool used for tracking source IP addresses (generated by the sticky-address and source-track options) to 2000.

You can combine them:

```
set limit { states 20000, frags 20000, src-nodes 2000 }
```

set optimization

Optimize the engine for one of the following network environments:

- normal — a normal network environment. Suitable for almost all networks.
- high-latency — a high-latency environment (such as a satellite connection).

- satellite — alias for high-latency.
- aggressive — aggressively expire connections. This can greatly reduce the memory usage of the firewall, at the cost of dropping idle connections early.
- conservative — extremely conservative settings. Avoid dropping legitimate connections at the expense of greater memory use (possibly much greater on a busy network) and slightly increased processor utilization.

For example:

```
set optimization aggressive
```

set block-policy

Set the default behavior for the packet block action:

- drop — the packet is silently dropped.
- return — a TCP RST is returned for blocked TCP packets, an ICMP UNREACHABLE is returned for blocked UDP packets, and all other packets are silently dropped.

For example:

```
set block-policy return
```

set state-policy

Set the default behavior for states:

- if-bound — states are bound to interface.
- group-bound — states are bound to interface group (e.g. ppp).
- floating — states can match packets on any interfaces (the default).

For example:

```
set state-policy if-bound
```

set require-order

By default, pfctl enforces an ordering of the statement types in the rule set to: options, normalization, queueing, translation, and filtering. Setting this option to no disables this enforcement.

There may be non-trivial and non-obvious implications to an out-of-order rule set. Consider carefully before disabling the order enforcement.

set fingerprints

Load fingerprints of known operating systems from the given file name. By default, fingerprints of known operating systems are automatically loaded from /etc/pf.os, but you can override the path with this option. For example:

```
set fingerprints "/etc/pf.os.devel"
```

Setting this option may leave a small period of time where the fingerprints referenced by the currently active rule set are inconsistent until the new rule set finishes loading.

set skip on *ifspec*

List the interfaces for which packets shouldn't be filtered. Packets passing in or out on such interfaces are passed as if pf were disabled, i.e. pf doesn't process them in any way. This can be useful on loopback and other virtual interfaces, when packet filtering isn't desired and can have unexpected effects. For example:

```
set skip on lo0
```

set debug

Set the debug level to one of the following:

- none — don't generate debug messages.
- urgent — generate debug messages only for serious errors.
- misc — generate debug messages for various errors.
- loud — generate debug messages for common conditions.

Traffic normalization

Traffic normalization is used to sanitize packet content in such a way that there are no ambiguities in packet interpretation on the receiving side. The normalizer does IP fragment reassembly to prevent attacks that confuse intrusion detection systems by sending overlapping IP fragments. Packet normalization is invoked with the scrub directive, which has the following options:

no-df

Clear the dont-fragment bit from a matching IP packet. Some operating systems are known to generate fragmented packets with the dont-fragment bit set. This is particularly true with NFS. Scrub will drop such fragmented dont-fragment packets unless you specify no-df.

Unfortunately some operating systems also generate their dont-fragment packets with a IP identification field of zero. Clearing the dont-fragment bit on packets with a zero IP ID may cause deleterious results if an upstream router later fragments the packet. Using the random-id modifier (see below) is recommended in combination with the no-df modifier to ensure unique IP identifiers.

min-ttl *number*

Enforce a minimum TTL for matching IP packets.

max-mss *number*

Enforce a maximum MSS for matching TCP packets.

random-id

Replace the IP identification field with random values to compensate for predictable values generated by many hosts. This option applies only to packets that aren't fragmented after the optional fragment reassembly.

fragment reassemble

Using scrub rules, fragments can be reassembled by normalization. In this case, fragments are buffered until they form a complete packet, and only the completed packet is passed on to the filter. The advantage is that filter rules have to deal only with complete packets, and can ignore fragments. The drawback of caching fragments is the additional memory cost. But the full reassembly method is the only method that currently works with NAT. This is the default behavior of a scrub rule if no fragmentation modifier is supplied.

fragment crop

The default fragment reassembly method is expensive, hence the option to crop is provided. In this case, pf tracks the fragments and caches a small range descriptor. Duplicate fragments are dropped, and overlaps are cropped. Thus data will occur only once on the wire with ambiguities resolving to the first occurrence. Unlike the fragment reassemble modifier, fragments aren't buffered; they're passed as soon as they're received. The fragment crop reassembly mechanism doesn't yet work with NAT.

fragment drop-ovl

This option is similar to the fragment crop modifier, except that all overlapping or duplicate fragments are dropped, and all further corresponding fragments are dropped as well.

reassemble tcp

Statefully normalize TCP connections. The scrub reassemble tcp rules may not have the direction (in/out) specified. The reassemble tcp performs the following normalizations:

- ttl — neither side of the connection is allowed to reduce their IP TTL. An attacker may send a packet such that it reaches the firewall, affects the firewall state, and expires before reaching the destination host. The reassemble tcp command raises the TTL of all packets back up to the highest value seen on the connection.
- timestamp modulation — modern TCP stacks send a timestamp on every TCP packet and echo the other endpoint's timestamp back to them. Many operating systems will merely start the timestamp at zero when first booted, and increment it several times a second. The uptime of the host can be deduced by reading the timestamp and multiplying it by a constant. Also observing several different timestamps can be used to count hosts behind a NAT device. And spoofing TCP packets into a connection requires knowing or guessing valid timestamps. Timestamps merely need to be monotonically increasing and not derived off a guessable base time.
  The reassemble tcp command will cause scrub to modulate the TCP timestamps with a random number.
- extended PAWS checks — there's a problem with TCP on long fat pipes, in that a packet might get delayed for longer than it takes the connection to wrap its 32-bit sequence space. In such an occurrence, the old packet would be indistinguishable from a new packet and would be accepted as such. The solution to this is called PAWS: Protection Against Wrapped Sequence numbers. It protects against it by making sure the timestamp on each packet doesn't go backwards. Thereassemble tcp command also makes sure the timestamp on the packet doesn't go forward more than the RFC allows. By doing this, pf artificially extends the security of TCP sequence numbers by 10 to 18 bits when the host uses appropriately randomized timestamps, since a blind attacker would have to guess the timestamp as well.

For example:

```
scrub in on $ext_if all fragment reassemble
```

The no option prefixed to a scrub rule causes matching packets to remain unscrubbed, much in the same way as drop quick works in the packet filter (see below). This mechanism should be used when it is necessary to exclude specific packets from broader scrub rules.

Queueing

Packets can be assigned to queues for the purpose of bandwidth control. At least two declarations are required to configure queues, and later any packet filtering rule can reference the defined queues by name. During the filtering component of pf.conf, the last referenced queue name is where any packets from pass rules will be queued, while for block rules it specifies where any resulting ICMP or TCP RST packets should be queued. The scheduler defines the algorithm used to decide which packets get delayed, dropped, or sent out immediately.
The currently supported schedulers are:

cbq

Class Based Queueing. Queues attached to an interface build a tree, and thus each queue can have further child queues. Each queue can have a priority and a bandwidth assigned. Priority mainly controls the time packets take to get sent out, while bandwidth has primarily effects on throughput.
The cbq scheduler achieves both partitioning and sharing of link bandwidth by hierarchically structured classes. Each class has its own queue and is assigned its share of bandwidth. A child class can borrow bandwidth from its parent class, as long as excess bandwidth is available (see the option borrow, below).

priq

Priority Queueing. Queues are flat attached to the interface, and thus queues can't have further child queues. Each queue has a unique priority assigned, ranging from 0 to 15. Packets in the queue with the highest priority are processed first.

hfsc

Hierarchical Fair Service Curve. Queues attached to an interface build a tree, and thus each queue can have further child queues. Each queue can have a priority and a bandwidth assigned. Priority mainly controls the time packets take to get sent out, while bandwidth has primarily effects on throughput.

The hfsc scheduler supports both link-sharing and guaranteed realtime services. It employs a service-curve-based QoS model, and its unique feature is an ability to decouple delay and bandwidth allocation.

The interfaces on which queueing should be activated are declared using the altq on declaration, which has the following keywords:

interface

Enable queueing on the named interface.

scheduler

Specify which queueing scheduler to use. Currently supported values are:

- cbq — Class Based Queueing
- priq — Priority Queueing
- hfsc — Hierarchical Fair Service Curve

bandwidth *bw*

The maximum bit rate for all queues on an interface. You can specify the value as an absolute value or as a percentage of the interface bandwidth. When using an absolute value, you can use the suffixes b,Kb, Mb, and Gb to represent bits, kilobits, megabits, and gigabits per second, respectively. The value mustn't exceed the interface bandwidth. If you don't specify bandwidth, the interface bandwidth is used.

qlimit *limit*

The maximum number of packets held in the queue. The default is 50.

tbrsize *size*

Adjust the size, in bytes, of the token bucket regulator. If not specified, heuristics based on the interface bandwidth are used to determine the size.

queue *list*

Define a list of subqueues to create on an interface.

In the following example, the interface dc0 should queue up to 5 Mbit/s in four second-level queues using Class Based Queueing. Those four queues will be shown in a later example:

```
altq on dc0 cbq bandwidth 5Mb queue { std, http, mail, ssh }
```

Once interfaces are activated for queueing using the altq directive, you can define a sequence of queue directives. The name associated with a queue must match a queue defined in the altqdirective (e.g. mail), or — except for the priq scheduler — in a parent queue declaration. You can use the following keywords:

- on *interface* — the interface the queue operates on. If not given, the queue operates on all matching interfaces.
- bandwidth *bw* — the maximum bit rate to be processed by the queue. This value mustn't exceed the value of the parent queue, and you can specify it as an absolute value or a percentage of the parent queue's bandwidth. If you don't specify this keyword, the bandwidth defaults to 100% of the parent queue's bandwidth. The priq scheduler doesn't support bandwidth specification.
- priority *level* — between queues, a priority level can be set. For cbq and hfsc, the range is 0 to 7; for priq, the range is 0 to 15. The default for all is 1. PRIQ queues with a higher priority are always served first. CBQ and HFSC queues with a higher priority are preferred in the case of overload.
- qlimit *limit* — the maximum number of packets held in the queue. The default is 50.

The scheduler can get additional parameters with *scheduler*(*parameters*). The parameters are as follows:

- default — packets not matched by another queue are assigned to this one. Exactly one default queue is required.
- red — enable RED (Random Early Detection) on this queue. RED drops packets with a probability proportional to the average queue length.
- rio — enable RIO on this queue. RIO is RED with IN/OUT, so running RED two times more than RIO would achieve

the same effect. RIO isn't currently supported in the GENERIC kernel.

- ecn — enable ECN (Explicit Congestion Notification) on this queue. ECN implies RED.

The CBQ scheduler supports an additional option:

- borrow — the queue can borrow bandwidth from the parent.

The HFSC scheduler supports some additional options (*sc* is an acronym for service curve):

- realtime *sc* — the minimum required bandwidth for the queue.
- upperlimit *sc* — the maximum allowed bandwidth for the queue.
- linkshare *sc* — the bandwidth share of a backlogged queue.

The format for service curve specifications is (*m1*, *d*, *m2*). The *m2* variable controls the bandwidth assigned to the queue, while *m1* and *d* are optional and can be used to control the initial bandwidth assignment. For the first *d* milliseconds, the queue gets the bandwidth given as *m1*, and afterward, the value given in *m2*.

Furthermore, with CBQ and HFSC, child queues can be specified as in an altq declaration, thus building a tree of queues using a part of their parent's bandwidth.

Packets can be assigned to queues based on filter rules by using the queue keyword. Normally only one queue is specified; when a second one is specified, it's used instead for packets that have a TOS of lowdelay and for TCP ACKs with no data payload.

To continue the previous example, the examples below specify the four referenced queues, plus a few child queues. The queues may then be referenced by filtering rules (see "Packet filtering," below).

```
queue std bandwidth 10% cbq(default)

queue http bandwidth 60% priority 2 cbq(borrow red) \

       { employees, developers }

queue  developers bandwidth 75% cbq(borrow)

queue  employees bandwidth 15%

queue mail bandwidth 10% priority 0 cbq(borrow ecn)

queue ssh bandwidth 20% cbq(borrow) { ssh_interactive, ssh_bulk }

queue  ssh_interactive bandwidth 50% priority 7 cbq(borrow)

queue  ssh_bulk bandwidth 50% priority 0 cbq(borrow)



block return out on dc0 inet all queue std

pass out on dc0 inet proto tcp from $developerhosts to any port 80 \

       keep state queue developers

pass out on dc0 inet proto tcp from $employeehosts to any port 80 \

       keep state queue employees
```

```
pass out on dc0 inet proto tcp from any to any port 22 \

       keep state queue(ssh_bulk, ssh_interactive)

pass out on dc0 inet proto tcp from any to any port 25 \

       keep state queue mail
```

Translation

Translation rules modify either the source or destination address of the packets associated with a stateful connection. A stateful connection is automatically created to track packets matching such a rule as long as they aren't blocked by the filtering section of pf.conf. The translation engine modifies the specified address and/or port in the packet, recalculates IP, TCP and UDP checksums as necessary, and passes it to the packet filter for evaluation.

Since translation occurs before filtering the filter engine will see packets as they look after any addresses and ports have been translated. Filter rules will therefore have to filter based on the translated address and port number. Packets that match a translation rule are automatically passed only if the pass modifier is given; otherwise they're still subject to block and pass rules.

The state entry created permits pf to keep track of the original address for traffic associated with that state and correctly direct return traffic for that connection.

Various types of translation are possible with pf:

binat

Specifies a bidirectional mapping between an external IP netblock and an internal IP netblock.

nat

Specifies that IP addresses are to be changed as the packet traverses the given interface. This technique allows one or more IP addresses on the translating host to support network traffic for a larger range of machines on an “inside” network. Although in theory any IP address can be used on the inside, it is strongly recommended that one of the address ranges defined by RFC 1918 be used. These netblocks are:

- 10.0.0.0–10.255.255.255 (all of net 10, i.e. 10/8)
- 172.16.0.0–172.31.255.255 (i.e., 172.16/12)
- 192.168.0.0–192.168.255.255 (i.e., 192.168/16)

rdr

The packet is redirected to another destination and possibly a different port. The rdr rules can optionally specify port ranges instead of single ports. For example:

```
rdr ... port 2000:2999 -> ... port 4000
```

redirects ports 2000 to 2999 (inclusive) to port 4000. This command:

```
rdr ... port 2000:2999 -> ... port 4000:*
```

redirects port 2000 to 4000, 2001 to 4001, ..., 2999 to 4999.

In addition to modifying the address, some translation rules may modify source or destination ports for TCP or UDP connections; implicitly in the case of nat rules and explicitly in the case of rdr rules. Port numbers are never translated with

a binat rule.
For each packet processed by the translator, the translation rules are evaluated in sequential order, from first to last. The first matching rule decides what action is taken.

The no option prefixed to a translation rule causes packets to remain untranslated, much in the same way as drop quick works in the packet filter (see below). If no rule matches the packet, the packet is passed to the filter engine unmodified.
Translation rules apply only to packets that pass through the specified interface, and if no interface is specified, translation is applied to packets on all interfaces. For instance, redirecting port 80 on an external interface to an internal web server works only for connections originating from the outside. Connections to the address of the external interface from local hosts aren't redirected, since such packets don't actually pass through the external interface. Redirections can't reflect packets back through the interface they arrive on; they can only be redirected to hosts connected to different interfaces or to the firewall itself.

Note that redirecting external incoming connections to the loopback address, as in:

```
rdr on ne3 inet proto tcp to port 8025 -> 127.0.0.1 port 25
```

effectively allows an external host to connect to daemons bound solely to the loopback address, circumventing the traditional blocking of such connections on a real interface. Unless this effect is desired, any of the local non-loopback addresses should be used as redirection target instead, which allows external connections only to daemons bound to this address or not bound to any address.

See “Translation examples,” below.

Packet filtering

The pf pseudo-device can block and pass packets based on attributes of their layer-3 (see IP and IPv6in the Neutrino Library Reference) and layer-4 (see ICMP, ICMP6, TCP, and UDP) headers. In addition, packets may also be assigned to queues for the purpose of bandwidth control.
For each packet processed by the packet filter, the filter rules are evaluated in sequential order, from first to last. The last matching rule decides what action is taken.

You can use the following actions in the filter:

block

Block the packet. There are a number of ways in which a block rule can behave when blocking a packet. The default behavior is to drop packets silently, however you can override this or make it explicit either globally, by setting the block-policy option, or on a per-rule basis with one of the following options:

- drop — silently drop the packet.
- return-rst — this applies only to TCP packets, and issues a TCP RST that closes the connection.
- return-icmp, return-icmp6 — cause ICMP messages to be returned for packets that match the rule. By default, this is an ICMP UNREACHABLE message, however you can override this by specifying a message as a code or number.
- return — cause a TCP RST to be returned for TCP packets and an ICMP UNREACHABLE for UDP and other packets.

Options returning ICMP packets currently have no effect if pf operates on a bridge, as the code to support this feature hasn't yet been implemented.

pass

Pass the packet.

If no rule matches the packet, the default action is to pass it. To block everything by default and pass only packets that match explicit rules, use:

```
block all
```

as the first filter rule.

See "Filter examples," below.

Parameters

The rule parameters specify the packets to which a rule applies. A packet always comes in on, or goes out through, one interface. Most parameters are optional. If a parameter is specified, the rule applies only to packets with matching attributes. Certain parameters can be expressed as lists, in which casepfctl generates all needed rule combinations.

in or out

This rule applies to incoming or outgoing packets. If you specify neither in nor out, the rule matches packets in both directions.

log

In addition to the action specified, generate a log message. All packets for that connection are logged, unless the keep state, modulate state or synproxy state options are specified, in which case only the packet that establishes the state is logged. (See keep state, modulate state andsynproxy state below). The logged packets are sent to the pflog interface. This interface is monitored by the pflogd logging daemon, which dumps the logged packets to the file/var/log/pflog in pcap binary format.

log-all

Used with keep state, modulate state or synproxy state rules to force logging of all packets for a connection. As with log, packets are logged to pflog.

quick

If a packet matches a rule that has the quick option set, this rule is considered the last matching rule, and evaluation of subsequent rules is skipped.

on *interface*

This rule applies only to packets coming in on, or going out through, this particular interface. It's also possible to simply give the interface driver name, such as ppp or fxp, to make the rule match packets flowing through a group of interfaces.

af

This rule applies only to packets of this address family. Supported values are inet and inet6.

proto *protocol*

This rule applies only to packets of this protocol. Common protocols are ICMP, ICMP6, TCP, and UDP. For a list of all the protocol name-to-number mappings used by pfctl, see the file /etc/protocols.

from *source* port *source* os *source* to *dest* port *dest*

This rule applies only to packets with the specified source and destination addresses and ports. You can specify addresses in CIDR notation (matching netblocks), as symbolic host names or interface names, or as any of the following keywords:

- any — any address.
- route *label* — any address whose associated route has the label specified by *label*. See theROUTE protocol (in the Neutrino Library Reference) and the route utility.
- no-route — any address that isn't currently routable.
- *table* — any address that matches the given table.

Interface names can have modifiers appended:

- :network — translates to the network(s) attached to the interface.
- :broadcast — translates to the interface's broadcast address(es).
- :peer — translates to the point-to-point interface's peer address(es).
- :0 — don't include interface aliases.

Host names may also have the :0 option appended to restrict the name resolution to the first of each v4 and v6 address found.

Host-name resolution and interface-to-address translation are done when the rule set is loaded. When the address of an interface (or host name) changes (under DHCP or PPP, for instance), the rule set must be reloaded for the change to be reflected in io-pkt.

Surrounding the interface name (and optional modifiers) in parentheses changes this behavior: the rule is automatically updated whenever the interface changes its address. The rule set doesn't need to be reloaded. This is especially useful with nat.

You can specify ports either by number or by name. For example, port 80 can be specified as www. For a list of all port name-to-number mappings used by pfctl, see the file /etc/services.

You can use these operators to specify ports and ranges of ports:

Operator	Meaning
`=`	Equal to
`!=`	Unequal to
`<`	Less than
`<=`	Less than or equal to
`>`	Greater than
`>=`	Greater than or equal to
`:`	Range, including boundaries
`><`	Range, excluding boundaries
`<>`	Except range

The ><, <> and : operators are binary; they take two arguments. For instance:

This:	Means:
`port 2000:2004`	All ports ≥ 2000 and ≤ 2004; hence ports 2000, 2001, 2002, 2003 and 2004
`port 2000 >< 2004`	All ports > 2000 and < 2004; hence ports 2001, 2002 and 2003
`port 2000 <> 2004`	All ports < 2000 or > 2004; hence ports 1-1999 and 2005-65535

The operating system of the source host can be specified in the case of TCP rules with the OS modifier.

See “Operating system fingerprinting” for more information.

The host, port and OS specifications are optional, as in the following examples:

```
pass in all

pass in from any to any

pass in proto tcp from any port <= 1024 to any

pass in proto tcp from any to any port 25

pass in proto tcp from 10.0.0.0/8 port > 1024 \

       to ! 10.1.2.3 port != ssh

pass in proto tcp from any os "OpenBSD" flags S/SA

pass in proto tcp from route "DTAG"
```

all

This is equivalent to from any to any.

group *group*

This functionality isn't supported in this version of NetBSD.

user *user*

This rule applies only to packets of sockets owned by the specified user. For outgoing connections initiated from the firewall, this is the user that opened the connection. For incoming connections to the firewall itself, this is the user that listens on the destination port. For forwarded connections, where the firewall isn't a connection endpoint, the user and group are unknown.
All packets, both outgoing and incoming, of one connection are associated with the same user and group. Only TCP and UDP packets can be associated with users; for other protocols these parameters are ignored.

User and group refer to the effective (as opposed to the real) IDs, in case the socket is created by asetuid or setgid process. User and group IDs are stored when a socket is created; when a process creates a listening socket as root (for instance, by binding to a privileged port) and subsequently changes to another user ID (to drop privileges), the credentials will remain root.
You can specify user and group IDs as either numbers or names. The syntax is similar to the one for ports. The value unknown matches packets of forwarded connections. You can use unknown only with the operators = and !=. Other constructions, such as user >= unknown, are invalid. Forwarded packets with unknown user and group ID match only rules that explicitly compare against unknownwith the operators = or !=. For instance, user >= 0 doesn't match forwarded packets. The following example allows only selected users to open outgoing connections:

```
block out proto { tcp, udp } all

pass  out proto { tcp, udp } all \

       user { < 1000, dhartmei } keep state
```

flags *a*/*b* | /*b*

This rule applies only to TCP packets that have the flags *a* set out of set *b*. Flags not specified in *b* are ignored. The flags are: (F)IN, (S)YN, (R)ST, (P)USH, (A)CK, (U)RG, (E)CE, and C(W)R. For example:

This:	Means:
`flags S/S`	Flag SYN is set. The other flags are ignored.
`flags S/SA`	Out of SYN and ACK, exactly SYN may be set. SYN, SYN+PSH and SYN+RST match, but SYN+ACK, ACK and ACK+RST don't. This is more restrictive than the previous example.
`flags /SFRA`	If the first set isn't specified, it defaults to none. All of SYN, FIN, RST and ACK must be unset.

icmp-type *type* code *code*
icmp6-type *type* code *code*

This rule applies only to ICMP or ICMPv6 packets with the specified type and code. Text names for ICMP types and codes are listed in the entries for ICMP and ICMP6 in the Neutrino Library Reference. This parameter is valid only for rules that cover protocols ICMP or ICMP6. The protocol and the ICMP type indicator (icmp-type or icmp6-type) must match.

tos *string* | *number*

This rule applies to packets with the specified TOS bits set. TOS may be given as one of lowdelay,throughput, reliability, or as either hexadecimal or decimal. For example, the following rules are identical:

- pass all tos lowdelay
- pass all tos 0x10
- pass all tos 16

allow-opts

By default, packets that contain IP options are blocked. When you specify allow-opts for a pass rule, packets that pass the filter based on that rule (last matching) do so even if they contain IP options. For packets that match a state, the rule that initially created the state is used. The implicit pass rule that's used when a packet doesn't match any rules doesn't allow IP options.

label *string*

Add a label (name) to the rule, which you can use to identify the rule. For instance, pfctl -s labelsshows per-rule statistics for rules that have labels.

You can use the following macros in labels:

Macro	Meaning
`$if`	The interface
`$srcaddr`	The source IP address
`$dstaddr`	The destination IP address
`$srcport`	The source port specification
`$dstport`	The destination port specification
`$proto`	The protocol name
`$nr`	The rule number

For example:

```
ips = "{ 1.2.3.4, 1.2.3.5 }"

pass in proto tcp from any to $ips \

     port > 1023 label "$dstaddr:$dstport"
```

expands to:

```
pass in inet proto tcp from any to 1.2.3.4 \

     port > 1023 label "1.2.3.4:>1023"

pass in inet proto tcp from any to 1.2.3.5 \

     port > 1023 label "1.2.3.5:>1023"
```

The macro expansion for the label directive occurs only when the configuration file is parsed, not during runtime.

queue *queue* | (*queue*, *queue*)

Packets matching this rule are assigned to the specified queue. If you specify two queues, packets that have a TOS of lowdelay and TCP ACKs with no data payload are assigned to the second one. See “Queueing,” for setup details.

For example:

```
pass in proto tcp to port 25 queue mail
pass in proto tcp to port 22 queue(ssh_bulk, ssh_prio)
```

tag *string*

Packets matching this rule are tagged with the specified string. The tag acts as an internal marker that can be used to identify these packets later on. You can use this, for example, to provide trust between interfaces and to determine if packets have been processed by translation rules.
Tags are “sticky”, meaning that the packet is tagged even if the rule isn't the last matching rule. Further matching rules can replace the tag with a new one, but don't remove a previously applied tag. A packet is only ever assigned one tag at a time. Any pass rules that use the tag keyword must also use keep state, modulate state or synproxy state.
You can tag packets during nat, rdr, or binat rules in addition to filter rules. Tags take the same macros as labels (see above).

tagged *string*

Used with filter or translation rules to specify that packets must already be tagged with the given tag in order to match the rule. You can also do inverse tag matching by specifying the ! operator before thetagged keyword.

probability *number*

A probability attribute can be attached to a rule, with a value set between 0 and 1, bounds not included. In that case, the rule is honored using the given probability value only. For example, the following rule drops 20% of incoming ICMP packets:

```
block in proto icmp probability 20%
```

Routing

If a packet matches a rule with a route option set, the packet filter routes the packet according to the type of route option. When such a rule creates state, the route option is also applied to all packets matching the same connection.

fastroute

Do a normal route lookup to find the next hop for the packet.

route-to

Route the packet to the specified interface with an optional address for the next hop. When a route-to rule creates state, only packets that pass in the same direction as the filter rule specifies are routed in this way. Packets passing in the opposite direction (replies) aren't affected and are routed normally.

reply-to

Similar to route-to, but routes packets that pass in the opposite direction (replies) to the specified interface. Opposite direction is defined only in the context of a state entry, and reply-to is useful only in rules that create state. You can use it on systems with multiple external connections to route all outgoing packets of a connection through the interface the incoming connection arrived through (symmetric routing enforcement).

dup-to

Create a duplicate of the packet and route it like route-to. The original packet gets routed as it normally would.

Pool options

For nat and rdr rules, (as well as for the route-to, reply-to and dup-to rule options) for which there is a single redirection address that has a subnet mask smaller than 32 for IPv4 or 128 for IPv6 (more than one IP address), you can use a variety of different methods for assigning this address:

bitmask

Apply the network portion of the redirection address to the address to be modified (source with nat, destination with rdr).

random

Select an address at random within the defined block of addresses.

source-hash

Use a hash of the source address to determine the redirection address, ensuring that the redirection address is always the same for a given source. You can optionally specify a key after this keyword either in hexadecimal or as a string; by default, pfctl randomly generates a key for source-hashevery time the rule set is reloaded.

round-robin

Loop through the redirection address(es).

When more than one redirection address is specified, round-robin is the only permitted pool type.

static-port

With nat rules, prevent pf from modifying the source port on TCP and UDP packets.

Additionally, you can specify the sticky-address option to help ensure that multiple connections from the same source are mapped to the same redirection address. You can use this option with therandom and round-robin pool options. Note that by default, these associations are destroyed as soon as there are no longer states that refer to them; in order to make the mappings last beyond the lifetime of the states, increase the global options with set timeout source-track. See “Stateful tracking options” for more ways to control the source tracking.

Stateful inspection

The pf packet filter is *stateful*, which means it can track the state of a connection. Instead of passing all traffic to port 25, for instance, it's possible to pass only the initial packet, and then begin to keep state. Subsequent traffic will flow because the filter is aware of the connection.

If a packet matches a pass ... keep state rule, the filter creates a state for this connection and automatically lets pass all subsequent packets of that connection.

Before any rules are evaluated, the filter checks whether the packet matches any state. If it does, the packet is passed without evaluation of any rules.

States are removed after the connection is closed or has timed out.

This has several advantages:

•Comparing a packet to a state involves checking its sequence numbers. If the sequence numbers are outside the narrow windows of expected values, the packet is dropped. This prevents spoofing attacks, such as when an attacker sends packets with a fake source address/port but doesn't know the connection's sequence numbers.

•Looking up states is usually faster than evaluating rules. If there are 50 rules, all of them are evaluated sequentially in O(n). Even with 50000 states, only 16 comparisons are needed to match a state, since states are stored in a binary search tree that allows searches in O(log2 n).

For instance:

```
block all

pass out proto tcp from any to any flags S/SA keep state

pass in  proto tcp from any to any port 25 flags S/SA keep state
```

This rule set blocks everything by default. Only outgoing connections and incoming connections to port 25 are allowed. The initial packet of each connection has the SYN flag set, will be passed and creates state. All further packets of these connections are passed if they match a state.

By default, packets coming in and out of any interface can match a state, but it's also possible to change that behavior by assigning states to a single interface or a group of interfaces.

The default policy is specified by the state-policy global option, but you can adjust this on a per-rule basis by adding one of the if-bound, group-bound, or floating keywords to the keep state option. For example, if a rule is defined as:

```
pass out on ppp from any to 10.12/16 keep state (group-bound)
```

a state created on ppp0 would match packets an all PPP interfaces, but not packets flowing throughfxp0 or any other interface.
Keeping rules floating is the more flexible option when the firewall is in a dynamic routing environment. However, this has some security implications, since a state created by one trusted network could allow potentially hostile packets coming in from other interfaces.

Specifying flags S/SA restricts state creation to the initial SYN packet of the TCP handshake. You can also be less restrictive, and allow state creation from intermediate (non-SYN) packets. This causes pfto synchronize to existing connections, for instance if you flush the state table.
For UDP, which is stateless by nature, keep state creates state as well. UDP packets are matched to states using only host addresses and ports.
ICMP messages fall into two categories: ICMP error messages, which always refer to a TCP or UDP packet, are matched against the referred to connection. If you keep state on a TCP connection, and an ICMP source quench message referring to this TCP connection arrives, it will be matched to the right state and get passed.

For ICMP queries, keep state creates an ICMP state, and pf knows how to match ICMP replies to states. For example:

```
pass out inet proto icmp all icmp-type echoreq keep state
```

allows echo requests (such as those created by ping) out, creates state, and matches incoming echo replies correctly to states.

The `nat`, `binat`, and `rdr` rules implicitly create state for connections.

State modulation
Much of the security derived from TCP is attributable to how well the initial sequence numbers (ISNs) are chosen. Some popular stack implementations choose very poor ISNs, and thus are normally susceptible to ISN prediction exploits. By applying a modulate state rule to a TCP connection, pfcreates a high-quality random sequence number for each connection endpoint.
The modulate state directive implicitly keeps state on the rule and is applicable only to TCP connections. For instance:

```
block all

pass out proto tcp from any to any modulate state

pass in  proto tcp from any to any port 25 flags S/SA modulate state
```

There are some caveats associated with state modulation:

- You can't apply a modulate state rule to a pre-existing but unmodulated connection. Such an application would desynchronize TCP's strict sequencing between the two endpoints. Instead, pftreats the modulate state modifier as a keep state modifier, and the pre-existing connection is inferred without the protection conferred by modulation.
- The other caveat affects currently modulated states when the state table is lost (firewall reboot, flushing the state table, etc...). The pf packet filter can't infer a connection again after the state table flushes the connection's modulator. When the state is lost, the connection may be left dangling until the respective endpoints time out the connection. It's possible on a fast local network for the endpoints to start an ACK storm while trying to resynchronize after the loss of the modulator. Using aflags S/SA modifier on modulate state rules between fast networks is suggested to prevent ACK storms.

SYN proxy
By default, pf passes packets that are part of a TCP handshake between the endpoints. You can use the synproxy state option to cause pf itself to complete the handshake with the active endpoint, perform a handshake with the passive endpoint, and then forward packets between the endpoints.
No packets are sent to the passive endpoint before the active endpoint has completed the handshake, hence so-called SYN

floods with spoofed source addresses will not reach the passive endpoint, as the sender can't complete the handshake.

The proxy is transparent to both endpoints, they each see a single connection from/to the other endpoint. The pf packet filter chooses random initial sequence numbers for both handshakes. Once the handshakes are completed, the sequence number modulators (see previous section) are used to translate further packets of the connection. Hence, synproxy state includes modulate state andkeep state.
Rules with synproxy won't work if pf operates on a bridge.
Example:

```
pass in proto tcp from any to any port www flags S/SA synproxy state
```

Stateful tracking options
All of keep state, modulate state and synproxy state support the following options:

max *number*
: Limit the number of concurrent states the rule may create. When this limit is reached, further packets matching the rule that would create state are dropped, until existing states time out.

timeout seconds
: Change the timeout values used for states created by this rule. For a list of all valid timeout names, see “Options,” above.

You can specify multiple options, separated by commas:

```
pass in proto tcp from any to any \

        port www flags S/SA keep state \

        (max 100, source-track rule, max-src-nodes 75, \

        max-src-states 3, tcp.established 60, tcp.closing 5)
```

If you specify the source-track keyword, the number of states per source IP is tracked:

source-track rule
: The maximum number of states created by this rule is limited by the rule's max-src-nodes and max-src-state options. Only state entries created by this particular rule count toward the rule's limits.

source-track global
: The number of states created by all rules that use this option is limited. Each rule can specify differentmax-src-nodes and max-src-states options, however state entries created by any participating rule count towards each individual rule's limits.

You can set the following limits:

max-src-nodes *number*
: Limit the maximum number of source addresses that can simultaneously have state table entries.

max-src-states *number*
: Limit the maximum number of simultaneous state entries that a single source address can create with this rule. For stateful TCP connections, limits on established connections (those that have completed the TCP 3-way handshake) can also be enforced per source IP.

max-src-conn *number*
: Limit the maximum number of simultaneous TCP connections that have completed the 3-way handshake that a single host can make.

max-src-conn-rate *number / seconds*
: Limit the rate of new connections over a time interval. The connection rate is an approximation calculated as a moving average.

Because the 3-way handshake ensures that the source address isn't being spoofed, more aggressive action can be taken based on these limits. With the overload *table* state option, source IP addresses that hit either of the limits on established connections will be added to the named table. You can use this table in the rule set to block further activity from the offending host, redirect it to a tarpit process, or restrict its bandwidth.

The optional flush keyword kills all states created by the matching rule that originate from the host that exceeds these limits. The global modifier to the flush command kills all states originating from the offending host, regardless of which rule created the state.
For example, the following rules protect the web server against hosts making more than 100 connections in 10 seconds. Any host that connects faster than this rate will have its address added to the *bad_hosts* table and have all states originating from it flushed. Any new packets arriving from this host will be dropped unconditionally by the block rule.

```
block quick from bad_hosts

pass in on $ext_if proto tcp to $webserver port www flags S/SA keep state \

       (max-src-conn-rate 100/10, overload bad_hosts flush global)
```

Operating system fingerprinting

Passive OS Fingerprinting is a mechanism to inspect nuances of a TCP con nection's initial SYN packet and guess at the host's operating system. Unfortunately these nuances are easily spoofed by an attacker so the fingerprint isn't useful in making security decisions. But the fingerprint is typically accurate enough to make policy decisions upon.

The fingerprints may be specified by operating system class, by version, or by subtype/patchlevel. The class of an operating system is typically the vendor or genre and would be OpenBSD for the pf firewall itself. The version of the oldest available OpenBSD release on the main ftp site would be 2.6 and the fingerprint would be written

```
"OpenBSD 2.6"
```

The subtype of an operating system is typically used to describe the patchlevel if that patch led to changes in the TCP stack behavior. In the case of OpenBSD, the only subtype is for a fingerprint that was normalized by the no-df scrub option and would be specified as:

```
"OpenBSD 3.3 no-df"
```

Fingerprints for most popular operating systems are provided by /etc/pf.os. Once pf is running, you can get a complete list of known operating system finger listed by running:

```
# pfctl -so
```

Filter rules can enforce policy at any level of operating system specification assuming a fingerprint is present. Policy could limit traffic to approved operating systems or even ban traffic from hosts that aren't at the latest service pack.

You can also use the unknown class as the fingerprint that matches packets for which no operating system fingerprint is known.
Examples:

```
pass  out proto tcp from any os OpenBSD keep state

block out proto tcp from any os Doors

block out proto tcp from any os "Doors PT"
block out proto tcp from any os "Doors PT SP3"

block out from any os "unknown"

pass on lo0 proto tcp from any os "OpenBSD 3.3 lo0" keep state
```

Operating system fingerprinting is limited only to the TCP SYN packet. This means that it doesn't work on other protocols and doesn't match a currently established connection.

Operating system fingerprints are occasionally wrong. There are several problems:

- an attacker can trivially craft his packets to appear as any operating system he chooses
- an operating system patch could change the stack behavior, and no fingerprints will match it until the database is updated
- multiple operating systems may have the same fingerprint.

Blocking spoofed traffic

Spoofing is the faking of IP addresses, typically for malicious purposes. The antispoof directive expands to a set of filter rules that block all traffic with a source IP from the network(s) directly connected to the specified interface(s) from entering the system through any other interface. For example, the line:

```
antispoof for lo0
```

expands to:

```
block drop in on ! lo0 inet from 127.0.0.1/8 to any

block drop in on ! lo0 inet6 from ::1 to any
```

For non-loopback interfaces, there are additional rules to block incoming packets with a source IP address identical to the interface's IP(s). For example, assuming the interface wi0 has an IP address of10.0.0.1 and a netmask of 255.255.255.0, the line:

```
antispoof for wi0 inet
```

expands to:

```
block drop in on ! wi0 inet from 10.0.0.0/24 to any

block drop in inet from 10.0.0.1 to any
```

Rules created by the `antispoof` directive interfere with packets sent over loopback interfaces to local addresses. You should pass these explicitly.

Fragment handling

The size of IP datagrams (packets) can be significantly larger than the maximum transmission unit (MTU) of the network. In cases when it is necessary or more efficient to send such large packets, the large packet will be fragmented into many smaller packets that will each fit onto the wire. Unfortunately for a firewalling device, only the first logical fragment will contain the necessary header information for the subprotocol that allows pf to filter on things such as TCP ports or to perform NAT.
Besides the use of scrub rules as described in “Traffic normalization” above, there are three options for handling fragments in the packet filter.
One alternative is to filter individual fragments with filter rules. If no scrub rule applies to a fragment, it's passed to the filter. Filter rules with matching IP header parameters decide whether the fragment is passed or blocked, in the same way as complete packets are filtered. Without reassembly, fragments can be filtered based only on IP header fields (source/destination address, protocol), since subprotocol header fields aren't available (TCP/UDP port numbers, ICMP code/type). You can use the fragmentoption to restrict filter rules to apply only to fragments, but not complete packets. Filter rules without thefragment option still apply to fragments, if they only specify IP header fields. For instance, the rule:

```
pass in proto tcp from any to any port 80
```

never applies to a fragment, even if the fragment is part of a TCP packet with destination port 80, because without reassembly this information isn't available for each fragment. This also means that fragments can't create new or match existing state table entries, which makes stateful filtering and address translation (NAT, redirection) for fragments impossible.

It's also possible to reassemble only certain fragments by specifying source or destination addresses or protocols as parameters in scrub rules.

In most cases, the benefits of reassembly outweigh the additional memory cost, and it's recommended you use scrub rules to reassemble all fragments via the fragment reassemble modifier.
You can limit the memory allocated for fragment caching by using pfctl. Once this limit is reached, fragments that would have to be cached are dropped until other entries time out. You can also adjust the timeout value.
Currently, only IPv4 fragments are supported, and IPv6 fragments are blocked unconditionally.

Anchors

Besides the main rule set, pfctl can load rule sets into anchor attachment points. An anchor is a container that can hold rules, address tables, and other anchors.
An anchor has a name that specifies the path where you can use pfctl to access the anchor to perform operations on it, such as attaching child anchors to it or loading rules into it. Anchors may be nested, with components separated by slashes (/), similar to how file system hierarchies are laid out. The main rule set is actually the default anchor, so filter and translation rules, for example, may also be contained in any anchor.
An anchor can reference another anchor attachment point using the following kinds of rules:

nat-anchor *name*
: Evaluate the nat rules in the specified anchor.

rdr-anchor *name*
: Evaluate the rdr rules in the specified anchor.

binat-anchor *name*
: Evaluate the binat rules in the specified anchor.

anchor *name*
: Evaluate the filter rules in the specified anchor.

load anchor *name* from *file*
: Load the rules from the specified file into the anchor name.

When evaluation of the main rule set reaches an anchor rule, pf evaluates all rules specified in that anchor.
Matching filter and translation rules in anchors with the quick option are final and abort the evaluation of the rules in other anchors and the main rule set.
Anchor rules are evaluated relative to the anchor in which they are contained. For example, all anchor rules specified in the main rule set reference anchor attachment points underneath the main rule set, and anchor rules specified in a file loaded from a load anchor rule are attached under that anchor point.

Rules may be contained in anchor attachment points that don't contain any rules when the main rule set is loaded, and later such anchors can be manipulated through pfctl without reloading the main rule set or other anchors. For example:

```
ext_if = "kue0"

block on $ext_if all

anchor spam

pass out on $ext_if all keep state

pass in on $ext_if proto tcp from any \

to $ext_if port smtp keep state
```

blocks all packets on the external interface by default, then evaluates all rules in the anchor namedspam, and finally passes all outgoing connections and incoming connections to port 25.
The following command loads a single rule into the anchor, which blocks all packets from a specific address:

```
# echo "block in quick from 1.2.3.4 to any" | \

         pfctl -a spam -f -
```

The anchor can also be populated by adding a load anchor rule after the anchor rule:

```
anchor spam

load anchor spam from "/etc/pf-spam.conf"
```

When pfctl loads pf.conf, it also loads all the rules from the file /etc/pf-spam.conf into the anchor.
Optionally, anchor rules can specify the parameter's direction, interface, address family, protocol and source/destination address/port using the same syntax as filter rules. When parameters are used, the anchor rule is evaluated only for matching packets. This allows conditional evaluation of anchors, like:

```
block on $ext_if all

anchor spam proto tcp from any to any port smtp

pass out on $ext_if all keep state

pass in on $ext_if proto tcp from any to $ext_if port smtp keep state
```

The rules inside the anchor spam are evaluated only for TCP packets with destination port 25. Hence:

```
# echo "block in quick from 1.2.3.4 to any" | \

        pfctl -a spam -f -
```

blocks connections only from 1.2.3.4 to port 25.
Anchors may end with the asterisk (*) character, which signifies that all anchors attached at that point should be evaluated in the alphabetical ordering of their anchor name. For example:

```
anchor "spam/*"
```

evaluates each rule in each anchor attached to the spam anchor. Note that it evaluates only anchors that are directly attached to the spam anchor, and doesn't descend to evaluate anchors recursively.
Since anchors are evaluated relative to the anchor in which they are contained, there's a mechanism for accessing the parent and ancestor anchors of a given anchor. Similar to file system path name resolution, if the sequence .. appears as an anchor path component, the parent anchor of the current anchor in the path evaluation at that point becomes the new current anchor. As an example, consider the following:

```
# echo ' anchor "spam/allowed" ' | pfctl -f -

# echo -e ' anchor "../banned" \n pass' | \

         pfctl -a spam/allowed -f -
```

Evaluation of the main rule set leads into the spam/allowed anchor, which evaluates the rules in thespam/banned anchor, if any, before finally evaluating the pass rule.
Since the parser specification for anchor names is a string, any reference to an anchor name containing slash (/) characters requires double quote (") characters around the anchor name.

Translation examples

This example maps incoming requests on port 80 to port 8080, on which a daemon is running (because, for example, it isn't run as root, and therefore lacks permission to bind to port 80):

```
# use a macro for the interface name, so it can be changed easily

ext_if = "ne3"



# map daemon on 8080 to appear to be on 80

rdr on $ext_if proto tcp from any to any port 80 -> 127.0.0.1 port 8080
```

If the pass modifier is given, packets matching the translation rule are passed without inspecting the filter rules:

```
rdr pass on $ext_if proto tcp from any to any port 80 -> 127.0.0.1 \

       port 8080
```

In the example below, vlan12 is configured as 192.168.168.1; the machine translates all packets coming from 192.168.168.0/24 to 204.92.77.111 when they're going out any interface exceptvlan12. This has the net effect of making traffic from the 192.168.168.0/24 network appear as though it is the Internet routable address 204.92.77.111 to nodes behind any interface on the router except for the nodes on vlan12. (Thus, 192.168.168.1 can talk to the 192.168.168.0/24 nodes.)

```
nat on ! vlan12 from 192.168.168.0/24 to any -> 204.92.77.111
```

In the example below, the machine sits between a fake internal 144.19.74.* network, and a routable external IP of 204.92.77.100. The no nat rule excludes protocol AH from being translated:

```
# NO NAT

no nat on $ext_if proto ah from 144.19.74.0/24 to any

nat on $ext_if from 144.19.74.0/24 to any -> 204.92.77.100
```

In the example below, packets bound for one specific server, as well as those generated by the system administrators aren't proxied; all other connections are:

```
# NO RDR

no rdr on $int_if proto { tcp, udp } from any to $server port 80

no rdr on $int_if proto { tcp, udp } from $sysadmins to any port 80

rdr on $int_if proto { tcp, udp } from any to any port 80 -> 127.0.0.1 \

       port 80
```

This longer example uses both a NAT and a redirection. The external interface has the address157.161.48.183. On the internal interface, we are running ftp-proxy, listening for outbound ftpsessions captured to port 8021:

```
# NAT

# Translate outgoing packets' source addresses (any protocol).

# In this case, any address but the gateway's external address is mapped.

nat on $ext_if inet from ! ($ext_if) to any -> ($ext_if)



# NAT PROXYING

# Map outgoing packets' source port to an assigned proxy port instead of

# an arbitrary port.

# In this case, proxy outgoing isakmp with port 500 on the gateway.

nat on $ext_if inet proto udp from any port = isakmp to any -> ($ext_if) \

       port 500



# BINAT

# Translate outgoing packets' source address (any protocol).

# Translate incoming packets' destination address to an internal machine

# (bidirectional).

binat on $ext_if from 10.1.2.150 to any -> $ext_if
```

```
# RDR

# Translate incoming packets' destination addresses.

# As an example, redirect a TCP and UDP port to an internal machine.

rdr on $ext_if inet proto tcp from any to ($ext_if) port 8080 \

       -> 10.1.2.151 port 22

rdr on $ext_if inet proto udp from any to ($ext_if) port 8080 \

       -> 10.1.2.151 port 53



# RDR

# Translate outgoing ftp control connections to send them to localhost

# for proxying with ftp-proxy(8) running on port 8021.

rdr on $int_if proto tcp from any to any port 21 -> 127.0.0.1 port 8021
```

In this example, a NAT gateway is set up to translate internal addresses using a pool of public addresses (192.0.2.16/28) and to redirect incoming web server connections to a group of web servers on the internal network:

```
# NAT LOAD BALANCE

# Translate outgoing packets' source addresses using an address pool.

# A given source address is always translated to the same pool address by

# using the source-hash keyword.

nat on $ext_if inet from any to any -> 192.0.2.16/28 source-hash



# RDR ROUND ROBIN

# Translate incoming web server connections to a group of web servers on

# the internal network.

rdr on $ext_if proto tcp from any to any port 80 \
       -> { 10.1.2.155, 10.1.2.160, 10.1.2.161 } round-robin
```

Filter examples

```
# The external interface is kue0

# (157.161.48.183, the only routable address)

# and the private network is 10.0.0.0/8, for which we are doing NAT.



# use a macro for the interface name, so it can be changed easily

ext_if = "kue0"



# normalize all incoming traffic

scrub in on $ext_if all fragment reassemble



# block and log everything by default

block return log on $ext_if all



# block anything coming from source we have no back routes for

block in from no-route to any



# block and log outgoing packets that don't have our address as source,

# they are either spoofed or something is misconfigured (NAT disabled,

# for instance), we want to be nice and don't send out garbage.

block out log quick on $ext_if from ! 157.161.48.183 to any



# silently drop broadcasts (cable modem noise)

block in quick on $ext_if from any to 255.255.255.255
```

```
# block and log incoming packets from reserved address space and invalid

# addresses, they are either spoofed or misconfigured, we can't reply to

# them anyway (hence, no return-rst).

block in log quick on $ext_if from { 10.0.0.0/8, 172.16.0.0/12, \

       192.168.0.0/16, 255.255.255.255/32 } to any



# ICMP



# pass out/in certain ICMP queries and keep state (ping)

# state matching is done on host addresses and ICMP ID (not type/code),

# so replies (like 0/0 for 8/0) will match queries

# ICMP error messages (which always refer to a TCP/UDP packet) are

# handled by the TCP/UDP states

pass on $ext_if inet proto icmp all icmp-type 8 code 0 keep state



# UDP



# pass out all UDP connections and keep state

pass out on $ext_if proto udp all keep state



# pass in certain UDP connections and keep state (DNS)

pass in on $ext_if proto udp from any to any port domain keep state



# TCP
# pass out all TCP connections and modulate state

pass out on $ext_if proto tcp all modulate state


# pass in certain TCP connections and keep state (SSH, SMTP, DNS, IDENT)

pass in on $ext_if proto tcp from any to any port { ssh, smtp, domain, \

       auth } flags S/SA keep state


# pass in data mode connections for ftp-proxy running on this host.

# (see ftp-proxy(8) for details)

pass in on $ext_if proto tcp from any to 157.161.48.183 port >= 49152 \

       flags S/SA keep state


# Don't allow Windows 9x SMTP connections since they are typically

# a viral worm. Alternately we could limit these OSes to 1 connection each.

block in on $ext_if proto tcp from any os {"Windows 95", "Windows 98"} \

       to any port smtp


# Packet Tagging


# three interfaces: $int_if, $ext_if, and $wifi_if (wireless). NAT is

# being done on $ext_if for all outgoing packets. tag packets in on

# $int_if and pass those tagged packets out on $ext_if.  all other

# outgoing packets (i.e., packets from the wireless network) are only

# permitted to access port 80.
```

```
pass in on $int_if from any to any tag INTNET keep state

pass in on $wifi_if from any to any keep state



block out on $ext_if from any to any

pass out quick on $ext_if tagged INTNET keep state

pass out on $ext_if proto tcp from any to any port 80 keep state
```

Grammar

The syntax for pf.conf in BNF is as follows:

```
line           = ( option | pf-rule | nat-rule | binat-rule | rdr-rule |

                   antispoof-rule | altq-rule | queue-rule | anchor-rule |

                   trans-anchors | load-anchors | table-rule )



option         = "set" ( [ "timeout" ( timeout | "{" timeout-list "}" ) ] |

                   [ "optimization" [ "default" | "normal" |

                   "high-latency" | "satellite" |

                   "aggressive" | "conservative" ] ]

                   [ "limit" ( limit-item | "{" limit-list "}" ) ] |

                   [ "loginterface" ( interface-name | "none" ) ] |

                   [ "block-policy" ( "drop" | "return" ) ] |

                   [ "state-policy" ( "if-bound" | "group-bound" |

                   "floating" ) ]

                   [ "require-order" ( "yes" | "no" ) ]

                   [ "fingerprints" filename ] |

                   [ "debug" ( "none" | "urgent" | "misc" | "loud" ) ] )
```

```
pf-rule        = action [ ( "in" | "out" ) ]
                 [ "log" | "log-all" ] [ "quick" ]
                 [ "on" ifspec ] [ route ] [ af ] [ protospec ]
                 hosts [ filteropt-list ]

filteropt-list = filteropt-list filteropt | filteropt
filteropt      = user | flags | icmp-type | icmp6-type | tos |
                 ( "keep" | "modulate" | "synproxy" ) "state"
                 [ "(" state-opts ")" ] |
                 "fragment" | "no-df" | "min-ttl" number |
                 "max-mss" number | "random-id" | "reassemble tcp" |
                 fragmentation | "allow-opts" |
                 "label" string | "tag" string | [ ! ] "tagged" string
                 "queue" ( string | "(" string [ [ "," ] string ] ")" ) |
                 "probability" number"%"

nat-rule       = [ "no" ] "nat" [ "pass" ] [ "on" ifspec ] [ af ]
                 [ protospec ] hosts [ "tag" string ] [ "tagged" string ]
                 [ "->" ( redirhost | "{" redirhost-list "}" )
                 [ portspec ] [ pooltype ] [ "static-port" ] ]

binat-rule     = [ "no" ] "binat" [ "pass" ] [ "on" interface-name ]
                 [ af ] [ "proto" ( proto-name | proto-number ) ]
                 "from" address [ "/" mask-bits ] "to" ipspec
```

```
                   [ "tag" string ] [ "tagged" string ]
                   [ "->" address [ "/" mask-bits ] ]

rdr-rule       = [ "no" ] "rdr" [ "pass" ] [ "on" ifspec ] [ af ]
                   [ protospec ] hosts [ "tag" string ] [ "tagged" string ]
                   [ "->" ( redirhost | "{" redirhost-list "}" )
                   [ portspec ] [ pooltype ] ]

antispoof-rule = "antispoof" [ "log" ] [ "quick" ]
                   "for" ( interface-name | "{" interface-list "}" )
                   [ af ] [ "label" string ]

table-rule     = "table" "<" string ">" [ tableopts-list ]
tableopts-list = tableopts-list tableopts | tableopts
tableopts      = "persist" | "const" | "file" string |
                   "{" [ tableaddr-list ] "}"
tableaddr-list = tableaddr-list [ "," ] tableaddr-spec | tableaddr-spec
tableaddr-spec = [ "!" ] tableaddr [ "/" mask-bits ]
tableaddr      = hostname | ipv4-dotted-quad | ipv6-coloned-hex |
                   interface-name | "self"

altq-rule      = "altq on" interface-name queueopts-list
                   "queue" subqueue
queue-rule     = "queue" string [ "on" interface-name ] queueopts-list
                   subqueue
```

```
anchor-rule    = "anchor" string [ ( "in" | "out" ) ] [ "on" ifspec ]
                   [ af ] [ "proto" ] [ protospec ] [ hosts ]

trans-anchors  = ( "nat-anchor" | "rdr-anchor" | "binat-anchor" ) string
                   [ "on" ifspec ] [ af ] [ "proto" ] [ protospec ] [ hosts ]

load-anchor    = "load anchor" string "from" filename

queueopts-list = queueopts-list queueopts | queueopts
queueopts      = [ "bandwidth" bandwidth-spec ] |
                   [ "qlimit" number ] | [ "tbrsize" number ] |
                   [ "priority" number ] | [ schedulers ]
schedulers     = ( cbq-def | priq-def | hfsc-def )
bandwidth-spec = "number" ( "b" | "Kb" | "Mb" | "Gb" | "%" )

action         = "pass" | "block" [ return ] | [ "no" ] "scrub"
return         = "drop" | "return" | "return-rst" [ "( ttl" number ")" ] |
                   "return-icmp" [ "(" icmpcode [ [ "," ] icmp6code ] ")" ] |
                   "return-icmp6" [ "(" icmp6code ")" ]
icmpcode       = ( icmp-code-name | icmp-code-number )
icmp6code      = ( icmp6-code-name | icmp6-code-number )

ifspec         = ( [ "!" ] interface-name ) | "{" interface-list "}"
interface-list = [ "!" ] interface-name [ [ "," ] interface-list ]
```

```
route          = "fastroute" |
                   ( "route-to" | "reply-to" | "dup-to" )
                   ( routehost | "{" routehost-list "}" )
                   [ pooltype ]
af             = "inet" | "inet6"


protospec      = "proto" ( proto-name | proto-number |
                   "{" proto-list "}" )
proto-list     = ( proto-name | proto-number ) [ [ "," ] proto-list ]


hosts          = "all" |
                   "from" ( "any" | "no-route" | "self" | host |
                   "{" host-list "}" | "route" string ) [ port ] [ os ]
                   "to"   ( "any" | "no-route" | "self" | host |
                   "{" host-list "}" | "route" string ) [ port ]


ipspec         = "any" | host | "{" host-list "}"
host           = [ "!" ] ( address [ "/" mask-bits ] | "<" string ">" )
redirhost      = address [ "/" mask-bits ]
routehost      = ( interface-name [ address [ "/" mask-bits ] ] )
address        = ( interface-name | "(" interface-name ")" | hostname |
                   ipv4-dotted-quad | ipv6-coloned-hex )
host-list      = host [ [ "," ] host-list ]
redirhost-list = redirhost [ [ "," ] redirhost-list ]
routehost-list = routehost [ [ "," ] routehost-list ]
```

```
port           = "port" ( unary-op | binary-op | "{" op-list "}" )

portspec       = "port" ( number | name ) [ ":" ( "*" | number | name ) ]

os             = "os"  ( os-name | "{" os-list "}" )

user           = "user" ( unary-op | binary-op | "{" op-list "}" )


unary-op       = [ "=" | "!=" | "<" | "<=" | ">" | ">=" ]

                   ( name | number )

binary-op      = number ( "<>" | "><" | ":" ) number

op-list        = ( unary-op | binary-op ) [ [ "," ] op-list ]


os-name        = operating-system-name

os-list        = os-name [ [ "," ] os-list ]


flags          = "flags" [ flag-set ] "/" flag-set

flag-set       = [ "F" ] [ "S" ] [ "R" ] [ "P" ] [ "A" ] [ "U" ] [ "E" ]

                   [ "W" ]


icmp-type      = "icmp-type" ( icmp-type-code | "{" icmp-list "}" )

icmp6-type     = "icmp6-type" ( icmp-type-code | "{" icmp-list "}" )

icmp-type-code = ( icmp-type-name | icmp-type-number )

                   [ "code" ( icmp-code-name | icmp-code-number ) ]

icmp-list      = icmp-type-code [ [ "," ] icmp-list ]


tos            = "tos" ( "lowdelay" | "throughput" | "reliability" |
                   [ "0x" ] number )



state-opts     = state-opt [ [ "," ] state-opts ]

state-opt      = ( "max" number | timeout |

                   "source-track" [ ( "rule" | "global" ) ] |

                   "max-src-nodes" number | "max-src-states" number |

                   "max-src-conn" number |

                   "max-src-conn-rate" number "/" number |

                   "overload" "<" string ">" [ "flush" ] |

                   "if-bound" | "group-bound" | "floating" )



fragmentation  = [ "fragment reassemble" | "fragment crop" |

                   "fragment drop-ovl" ]



timeout-list   = timeout [ [ "," ] timeout-list ]

timeout        = ( "tcp.first" | "tcp.opening" | "tcp.established" |

                   "tcp.closing" | "tcp.finwait" | "tcp.closed" |

                   "udp.first" | "udp.single" | "udp.multiple" |

                   "icmp.first" | "icmp.error" |

                   "other.first" | "other.single" | "other.multiple" |

                   "frag" | "interval" | "src.track" |

                   "adaptive.start" | "adaptive.end" ) number



limit-list     = limit-item [ [ "," ] limit-list ]

limit-item     = ( "states" | "frags" | "src-nodes" ) number
```

```
pooltype       = ( "bitmask" | "random" |
                   "source-hash" [ ( hex-key | string-key ) ] |
                   "round-robin" ) [ sticky-address ]

subqueue       = string | "{" queue-list "}"
queue-list     = string [ [ "," ] string ]
cbq-def        = "cbq" [ "(" cbq-opt [ [ "," ] cbq-opt ] ")" ]
priq-def       = "priq" [ "(" priq-opt [ [ "," ] priq-opt ] ")" ]
hfsc-def       = "hfsc" [ "(" hfsc-opt [ [ "," ] hfsc-opt ] ")" ]
cbq-opt        = ( "default" | "borrow" | "red" | "ecn" | "rio" )
priq-opt       = ( "default" | "red" | "ecn" | "rio" )
hfsc-opt       = ( "default" | "red" | "ecn" | "rio" |
                   linkshare-sc | realtime-sc | upperlimit-sc )
linkshare-sc   = "linkshare" sc-spec
realtime-sc    = "realtime" sc-spec
upperlimit-sc  = "upperlimit" sc-spec
sc-spec        = ( bandwidth-spec |
                   "(" bandwidth-spec number bandwidth-spec ")" )
```

Associated files

/etc/hosts
    Host name database.

/etc/pf.os
    Default location of OS fingerprints.

/etc/protocols
    Protocol name database.

/etc/services
    Service name database.

/usr/share/examples/pf
    Examples of rule sets.

See also:

/etc/hosts, pf, /etc/pf.os, pfctl, /etc/protocols, route, /etc/services
ICMP, ICMP6, IP, IP6, ROUTE, TCP, UDP in the Neutrino Library Reference