Language: en [teams]
de fr it nl pl pt ru

[up to FAQ] [Next: Getting Started]

PF: The OpenBSD Packet Filter

Table of Contents

- Basic Configuration

- Advanced Configuration

- Additional Topics

- Example Rulesets

Packet Filter (from here on referred to as PF) is OpenBSD's system for filtering TCP/IP traffic and doing Network Address Translation. PF is also capable of normalizing and conditioning TCP/IP traffic and providing bandwidth control and packet prioritization. PF has been a part of the GENERIC OpenBSD kernel since OpenBSD 3.0. Previous OpenBSD releases used a different firewall/NAT package which is no longer supported.

PF was originally developed by Daniel Hartmeier and is now maintained and developed by the entire OpenBSD team.

This set of documents, also available in PDF format, is intended as a general introduction to the PF system as run on OpenBSD. Even if it covers all of PF's major features, it is only intended to be used as a supplement to the man pages, and not as a replacement for them.

For a complete and in-depth view of what PF can do, please start by reading the pf(4) man page.

As with the rest of the FAQ, this set of documents is focused on users of OpenBSD 4.7. As PF is always growing and developing, there are changes and enhancements between the 4.7-release version and the version in OpenBSD-current as well as differences between 4.7 and earlier versions. The reader is advised to see the man pages for the version of OpenBSD they are currently working with. **In particular, there are significant differences between 4.6 and 4.7.**

[up to FAQ] [Next: Getting Started]

---

www@openbsd.org

$OpenBSD: index.html,v 1.47 2010/05/19 13:25:16 sthen Exp $

[Contents] [Next: Lists and Macros]

PF: Getting Started

Table of Contents

Activation

PF is enabled by default. If you wish to disable it on boot, add the line

```
pf=NO
```

to the file /etc/rc.conf.local and reboot your system to have it take effect.

You can also manually activate and deactivate PF by using the pfctl(8) program:

```
# pfctl -e
# pfctl -d
```

to enable and disable, respectively. Note that this just enables or disables PF, it doesn't actually load a ruleset. The ruleset must be loaded separately, either before or after PF is enabled.

Configuration

PF reads its configuration rules from `/etc/pf.conf` at boot time, as loaded by the rc scripts. Note that while `/etc/pf.conf` is the default and is loaded by the system rc scripts, it is just a text file loaded and interpreted by pfctl(8) and inserted into pf(4). For some applications, other rulesets may be loaded from other files after boot. As with any well designed Unix application, PF offers great flexibility.

The `pf.conf` file has five parts:

- **Macros:** User-defined variables that can hold IP addresses, interface names, etc.
- **Tables:** A structure used to hold lists of IP addresses.
- **Options:** Various options to control how PF works.
- **Queueing:** Provides bandwidth control and packet prioritization.
- **Filter Rules:** Allows the selective filtering or blocking of packets as they pass through any of the interfaces.
  Filter rules can be given parameters to specify **network address translation** (NAT) and packet redirection.

Blank lines are ignored, and lines beginning with # are treated as comments.

Control

After boot, PF operation can be managed using the pfctl(8) program. Some example commands are:

```
# pfctl -f /etc/pf.conf     Load the pf.conf file
# pfctl -nf /etc/pf.conf    Parse the file, but don't load it

# pfctl -sr                 Show the current ruleset
# pfctl -ss                 Show the current state table
# pfctl -si                 Show filter stats and counters
# pfctl -sa                 Show EVERYTHING it can show
```

For a complete list of commands, please see the pfctl(8) man page.

[Contents] [Next: Lists and Macros]

www@openbsd.org

$OpenBSD: config.html,v 1.31 2010/05/19 13:25:16 sthen Exp $

PF: Lists and Macros

Table of Contents

Lists

A list allows the specification of multiple similar criteria within a rule. For example, multiple protocols, port numbers, addresses, etc. So, instead of writing one filter rule for each IP address that needs to be blocked, one rule can be written by specifying the IP addresses in a list. Lists are defined by specifying items within `{ }` brackets.

When pfctl(8) encounters a list during loading of the ruleset, it creates multiple rules, one for each item in the list. For example:

```
block out on fxp0 from { 192.168.0.1, 10.5.32.6 } to any
```

gets expanded to:

```
block out on fxp0 from 192.168.0.1 to any
block out on fxp0 from 10.5.32.6 to any
```

Multiple lists can be specified within a rule:

```
match in on fxp0 proto tcp to port { 22 80 } rdr-to
192.168.0.6
block out on fxp0 proto { tcp udp } from { 192.168.0.1, \
   10.5.32.6 } to any port { ssh telnet }
```

Note that the commas between list items are optional.

Lists can also contain nested lists:

```
trusted = "{ 192.168.1.2 192.168.5.36 }"
pass in inet proto tcp from { 10.10.0.0/24 $trusted } to
port 22
```

Beware of constructs like the following, dubbed "negated lists", which are a common mistake:

```
pass in on fxp0 from { 10.0.0.0/8, !10.1.2.3 }
```

While the intended meaning is usually to match "any address within 10.0.0.0/8, except for 10.1.2.3", the rule expands to:

```
pass in on fxp0 from 10.0.0.0/8
pass in on fxp0 from !10.1.2.3
```

which matches any possible address. Instead, a table should be used.

Macros

Macros are user-defined variables that can hold IP addresses, port numbers, interface names, etc. Macros can reduce the complexity of a PF ruleset and also make maintaining a ruleset much easier.

Macro names must start with a letter and may contain letters, digits, and underscores. Macro names cannot be reserved words such as `pass`, `out`, or `queue`.

```
ext_if = "fxp0"

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

This creates a macro named `ext_if`. When a macro is referred to after it's been created, its name is preceded with a `$` character.

Macros can also expand to lists, such as:

```
friends = "{ 192.168.1.1, 10.0.2.5, 192.168.43.53 }"
```

Macros can be defined recursively. Since macros are not expanded within quotes the following syntax must be used:

```
    host1 = "192.168.1.1"
    host2 = "192.168.1.2"
    all_hosts = "{" $host1 $host2 "}"
```

The macro `$all_hosts` now expands to 192.168.1.1, 192.168.1.2.

[Previous: Getting Started] [Contents] [Next: Tables]

---

www@openbsd.org

$OpenBSD: macros.html,v 1.26 2010/05/19 13:25:16 sthen Exp $

[Previous: Lists and Macros] [Contents] [Next: Packet Filtering]

PF: Tables

Table of Contents

Introduction

A table is used to hold a group of IPv4 and/or IPv6 addresses. Lookups against a table are very fast and consume less memory and processor time than lists. For this reason, a table is ideal for holding a large group of addresses as the lookup time on a table holding 50,000 addresses is only slightly more than for one holding 50 addresses. Tables can be used in the following ways:

- source and/or destination address in rules.
- translation and redirection addresses `nat-to` and `rdr-to` rule options, respectively.
- destination address in `route-to`, `reply-to`, and `dup-to` rule options.

Tables are created either in `pf.conf` or by using pfctl(8).

Configuration

In `pf.conf`, tables are created using the `table` directive. The following attributes may be specified for each table:

- `const` - the contents of the table cannot be changed once the table is created. When this attribute is not specified, pfctl(8) may be used to add or remove addresses from the table at any time, even when running with a securelevel(7) of two or greater.

- persist - causes the kernel to keep the table in memory even when no rules refer to it. Without this attribute, the kernel will automatically remove the table when the last rule referencing it is flushed.

Example:

```
table <goodguys> { 192.0.2.0/24 }
table <rfc1918> const { 192.168.0.0/16, 172.16.0.0/12, \
   10.0.0.0/8 }
table <spammers> persist

block in on fxp0 from { <rfc1918>, <spammers> } to any
pass  in on fxp0 from <goodguys> to any
```

Addresses can also be specified using the negation (or "not") modifier such as:

```
table <goodguys> { 192.0.2.0/24, !192.0.2.5 }
```

The goodguys table will now match all addresses in the 192.0.2.0/24 network except for 192.0.2.5.

Note that table names are always enclosed in < > angled brackets.

Tables can also be populated from text files containing a list of IP addresses and networks:

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

block in on fxp0 from <spammers> to any
```

The file /etc/spammers would contain a list of IP addresses and/or CIDR network blocks, one per line. Any line beginning with # is treated as a comment and ignored.

Manipulating with pfctl

Tables can be manipulated on the fly by using pfctl(8). For instance, to add entries to the <spammers> table created above:

```
# pfctl -t spammers -T add 218.70.0.0/16
```

This will also create the <spammers> table if it doesn't already exist. To list the addresses in a table:

```
# pfctl -t spammers -T show
```

The -v argument can also be used with -Tshow to display statistics for each table entry. To remove addresses from a table:

```
# pfctl -t spammers -T delete 218.70.0.0/16
```

For more information on manipulating tables with pfctl, please read the pfctl(8) manpage.

Specifying Addresses

In addition to being specified by IP address, hosts may also be specified by their hostname. When the hostname is resolved to an IP address, all resulting IPv4 and IPv6 addresses are placed into the table. IP addresses can also be entered into a table by specifying a valid interface name, interface group, or the self keyword. The table will then contain all IP addresses assigned to that interface or group, or to the machine (including loopback addresses), respectively.

One limitation when specifying addresses is that 0.0.0.0/0 and 0/0 will not work in tables. The alternative is to hard code that address or use a macro.

Address Matching

An address lookup against a table will return the most narrowly matching entry. This allows for the creation of tables such as:

```
table <goodguys> { 172.16.0.0/16, !172.16.1.0/24,
172.16.1.100 }

block in on dc0
pass  in on dc0 from <goodguys>
```

Any packet coming in through dc0 will have its source address matched against the table <goodguys>:

- 172.16.50.5 - narrowest match is 172.16.0.0/16; packet matches the table and will be passed
- 172.16.1.25 - narrowest match is !172.16.1.0/24; packet matches an entry in the table but that entry is negated (uses the "!" modifier); packet does not match the table and will be blocked
- 172.16.1.100 - exactly matches 172.16.1.100; packet matches the table and will be passed
- 10.1.4.55 - does not match the table and will be blocked

[Previous: Lists and Macros] [Contents] [Next: Packet Filtering]

---

www@openbsd.org

$OpenBSD: tables.html,v 1.27 2010/05/19 13:25:16 sthen Exp $

[Previous: Tables] [Contents] [Next: Network Address Translation]

PF: Packet Filtering

Table of Contents

Introduction

Packet filtering is the selective passing or blocking of data packets as they pass through a network interface. The criteria that pf(4) uses when inspecting packets are based on the Layer 3 (IPv4 and IPv6) and Layer 4 (TCP, UDP, ICMP, and ICMPv6) headers. The most often used criteria are source and destination address, source and destination port, and protocol.

Filter rules specify the criteria that a packet must match and the resulting action, either block or pass, that is taken when a match is found. Filter rules are evaluated in sequential order, first to last. Unless the packet matches a rule containing the `quick` keyword, the packet will be evaluated against *all* filter rules before the final action is taken. The last rule to match is the "winner" and will dictate what action to take on the packet. There is an implicit `pass all` at the beginning of a filtering ruleset meaning that if a packet does not match any filter rule the resulting action will be `pass`.

Rule Syntax

The general, *highly simplified* syntax for filter rules is:

```
action [direction] [log] [quick] [on interface] [af] [proto
protocol] \
   [from src_addr [port src_port]] [to dst_addr [port dst_port]] \
   [flags tcp_flags] [state]
```

`action`

The action to be taken for matching packets, either `pass` or `block`. The `pass` action will pass the packet back to the kernel for further processing while the `block` action will react based on the setting of the `block-policy` option. The default reaction may be overridden by specifying either `block drop` or `block return`.

`direction`

The direction the packet is moving on an interface, either `in` or `out`.

`log`

Specifies that the packet should be logged via pflogd(8). If the rule creates state then only the packet which establishes the state is logged. To log all packets regardless, use `log (all)`.

`quick`

If a packet matches a rule specifying `quick`, then that rule is considered the last matching rule and the specified *`action`* is taken.

`interface`

The name or group of the network interface that the packet is moving through. Interfaces can be added to arbitrary groups using the ifconfig(8) command. Several groups are also automatically created by the kernel:

- The `egress` group, which contains the interface(s) that holds the default route(s).
- Interface family group for cloned interfaces. For example: `ppp` or `carp`.

This would cause the rule to match for any packet traversing any `ppp` or `carp` interface, respectively.

`af`

The address family of the packet, either `inet` for IPv4 or `inet6` for IPv6. PF is usually able to determine this parameter based on the source and/or destination address(es).

`protocol`

The Layer 4 protocol of the packet:

- `tcp`
- `udp`
- `icmp`
- `icmp6`
- A valid protocol name from `/etc/protocols`
- A protocol number between 0 and 255
- A set of protocols using a list.

`src_addr`, *`dst_addr`*

The source/destination address in the IP header. Addresses can be specified as:

- A single IPv4 or IPv6 address.
- A CIDR network block.
- A fully qualified domain name that will be resolved via DNS when the ruleset is loaded. All

resulting IP addresses will be substituted into the rule.
  - The name of a network interface or group. Any IP addresses assigned to the interface will be substituted into the rule.
  - The name of a network interface followed by `/netmask` (i.e., `/24`). Each IP address on the interface is combined with the netmask to form a CIDR network block which is substituted into the rule.
  - The name of a network interface or group in parentheses `( )`. This tells PF to update the rule if the IP address(es) on the named interface change. This is useful on an interface that gets its IP address via DHCP or dial-up as the ruleset doesn't have to be reloaded each time the address changes.
  - The name of a network interface followed by any one of these modifiers:
    - `:network` - substitutes the CIDR network block (e.g., 192.168.0.0/24)
    - `:broadcast` - substitutes the network broadcast address (e.g., 192.168.0.255)
    - `:peer` - substitutes the peer's IP address on a point-to-point link

    In addition, the `:0` modifier can be appended to either an interface name or to any of the above modifiers to indicate that PF should not include aliased IP addresses in the substitution. These modifiers can also be used when the interface is contained in parentheses. Example: `fxp0:network:0`
  - A table.
  - The keyword `urpf-failed` can be used for the source address to indicate that it should be run through the uRPF check.
  - Any of the above but negated using the `!` ("not") modifier.
  - A set of addresses using a list.
  - The keyword `any` meaning all addresses
  - The keyword `all` which is short for `from any to any`.

`src_port`, *`dst_port`*

The source/destination port in the Layer 4 packet header. Ports can be specified as:

  - A number between 1 and 65535
  - A valid service name from `/etc/services`
  - A set of ports using a list
  - A range:
    - `!=` (not equal)
    - `<` (less than)
    - `>` (greater than)
    - `<=` (less than or equal)
    - `>=` (greater than or equal)
    - `><` (range)
    - `<>` (inverse range)

      The last two are binary operators (they take two arguments) and do not include the arguments in the range.
    - `:` (inclusive range)

      The inclusive range operator is also a binary operator and does include the arguments in the range.

`tcp_flags`

Specifies the flags that must be set in the TCP header when using `proto tcp`. Flags are specified as `flags` *`check/mask`*. For example: `flags S/SA` - this instructs PF to only look at the S and A (SYN and ACK) flags and to match if only the SYN flag is "on" (and is applied to all TCP rules by default). `flags any` instructs PF not to check flags.

`state`

Specifies whether state information is kept on packets matching this rule.

- `no state` - works with TCP, UDP, and ICMP. PF will not track this connection statefully. For TCP connections, `flags any` is usually also required.
- `keep state` - works with TCP, UDP, and ICMP. This option is the default for all filter rules.
- `modulate state` - works only with TCP. PF will generate strong Initial Sequence Numbers (ISNs) for packets matching this rule.
- `synproxy state` - proxies incoming TCP connections to help protect servers from spoofed TCP SYN floods. This option includes the functionality of `keep state` and `modulate state`.

Default Deny

The recommended practice when setting up a firewall is to take a "default deny" approach. That is, to deny *everything* and then selectively allow certain traffic through the firewall. This approach is recommended because it errs on the side of caution and also makes writing a ruleset easier.

To create a default deny filter policy, the first two filter rules should be:

```
block in  all
block out all
```

This will block all traffic on all interfaces in either direction from anywhere to anywhere.

Passing Traffic

Traffic must now be explicitly passed through the firewall or it will be dropped by the default deny policy. This is where packet criteria such as source/destination port, source/destination address, and protocol come into play. Whenever traffic is permitted to pass through the firewall the rule(s) should be written to be as restrictive as possible. This is to ensure that the intended traffic, and only the intended traffic, is permitted to pass.

Some examples:

```
# Pass traffic in on dc0 from the local network, 192.168.0.0/24,
# to the OpenBSD machine's IP address 192.168.0.1. Also, pass the
# return traffic out on dc0.
pass in  on dc0 from 192.168.0.0/24 to 192.168.0.1
pass out on dc0 from 192.168.0.1 to 192.168.0.0/24


# Pass TCP traffic in on fxp0 to the web server running on the
# OpenBSD machine. The interface name, fxp0, is used as the
# destination address so that packets will only match this rule if
# they're destined for the OpenBSD machine.
pass in on fxp0 proto tcp from any to fxp0 port www
```

The quick Keyword

As indicated earlier, each packet is evaluated against the filter ruleset from top to bottom. By default, the packet is marked for passage, which can be changed by any rule, and could be changed back and forth several times before the end of the filter rules. **The last matching rule "wins".** There is an exception to this: The `quick` option on a filtering rule has the effect of canceling any further rule processing and causes the specified action to be taken. Let's look at a couple examples:

Wrong:

```
block in on fxp0 proto tcp to port ssh
pass  in all
```

In this case, the `block` line may be evaluated, but will never have any effect, as it is then followed by a line which will pass everything.

Better:

```
block in quick on fxp0 proto tcp to port ssh
pass  in all
```

These rules are evaluated a little differently. If the `block` line is matched, due to the `quick` option, the packet will be blocked, and the rest of the ruleset will be ignored.

Keeping State

One of Packet Filter's important abilities is "keeping state" or "stateful inspection". Stateful inspection refers to PF's ability to track the state, or progress, of a network connection. By storing information about each connection in a state table, PF is able to quickly determine if a packet passing through the firewall belongs to an already established connection. If it does, it is passed through the firewall without going through ruleset evaluation.

Keeping state has many advantages including simpler rulesets and better packet filtering performance. PF is able to match packets moving in *either* direction to state table entries meaning that filter rules which pass returning traffic don't need to be written. And, since packets matching stateful connections don't go through ruleset evaluation, the time PF spends processing those packets can be greatly lessened.

When a rule creates state, the first packet matching the rule creates a "state" between the sender and receiver. Now, not only do packets going from the sender to receiver match the state entry and bypass ruleset evaluation, but so do the reply packets from receiver to sender.

All *pass* rules automatically create a state entry when a packet matches the rule. This can be explicitly disabled by using the `no state` option.

```
pass out on fxp0 proto tcp from any to any
```

This rule allows any outbound TCP traffic on the `fxp0` interface and also permits the reply traffic to pass back through the firewall. Keeping state significantly improves the performance of your firewall as state lookups are dramatically faster than running a packet through the filter rules.

The `modulate state` option works just like `keep state` except that it only applies to TCP packets. With `modulate state`, the Initial Sequence Number (ISN) of outgoing connections is randomized. This is useful for protecting connections initiated by certain operating systems that do a poor job of choosing ISNs. To allow simpler rulesets, the `modulate state` option can be used in rules that specify protocols other than TCP; in those cases, it is treated as `keep state`.

Keep state on outgoing TCP, UDP, and ICMP packets and modulate TCP ISNs:

```
pass out on fxp0 proto { tcp, udp, icmp } from any \
    to any modulate state
```

Another advantage of keeping state is that corresponding ICMP traffic will be passed through the firewall. For example, if a TCP connection passing through the firewall is being tracked statefully and an ICMP source-quench message referring to this TCP connection arrives, it will be matched to the appropriate state entry and passed through the firewall.

The scope of a state entry is controlled globally by the `state-policy` runtime option and on a per rule basis by the `if-bound` and `floating` state option keywords. These per rule keywords have the same meaning as when used with the `state-policy` option. Example:

```
pass out on fxp0 proto { tcp, udp, icmp } from any \
    to any modulate state (if-bound)
```

This rule would dictate that in order for packets to match the state entry, they must be transiting the `fxp0` interface.

Keeping State for UDP

One will sometimes hear it said that, "One can not create state with UDP as UDP is a stateless protocol!" While it is true that a UDP communication session does not have any concept of state (an explicit start and stop of communications), this does not have any impact on PF's ability to create state for a UDP session. In the case of protocols without "start" and "end" packets, PF simply keeps track of how long it has been since a matching packet has gone through. If the timeout is reached, the state is cleared. The timeout values can be set in the options section of the `pf.conf` file.

Stateful Tracking Options

Filter rules that create state entries can specify various options to control the behavior of the resulting state entry. The following options are available:

`max` *`number`*

Limit the maximum number of state entries the rule can create to *number*. If the maximum is reached, packets that would normally create state fail to match this rule until the number of existing states decreases below the limit.

`no state`

Prevents the rule from automatically creating a state entry.

`source-track`

This option enables the tracking of number of states created per source IP address. This option has two formats:

- `source-track rule` - The maximum number of states created by this rule is limited by the rule's `max-src-nodes` and `max-src-states` 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 different `max-src-nodes` and `max-src-states` options, however state entries created by any participating rule count towards each individual rule's limits.

The total number of source IP addresses tracked globally can be controlled via the `src-nodes` runtime option.

`max-src-nodes` *`number`*

When the `source-track` option is used, `max-src-nodes` will limit the number of source IP addresses that can simultaneously create state. This option can only be used with `source-track rule`.

`max-src-states` *`number`*

When the `source-track` option is used, `max-src-states` will limit the number of simultaneous state entries that can be created per source IP address. The scope of this limit (i.e., states created by this rule only or states created by all rules that use `source-track`) is dependent on the `source-track` option specified.

Options are specified inside parenthesis and immediately after one of the state keywords (`keep state`, `modulate state`, or `synproxy state`). Multiple options are separated by commas. In OpenBSD 4.1 and later, the `keep state` option became the implicit default for all filter rules. Despite this, when specifying stateful options, one of the state keywords must still be used in front of the options.

An example rule:

```
pass in on $ext_if proto tcp to $web_server \
    port www keep state \
    (max 200, source-track rule, max-src-nodes 100, max-src-states
3)
```

The rule above defines the following behavior:

- Limit the absolute maximum number of states that this rule can create to 200
- Enable source tracking; limit state creation based on states created by this rule only
- Limit the maximum number of nodes that can simultaneously create state to 100
- Limit the maximum number of simultaneous states per source IP to 3

A separate set of restrictions can be placed on stateful TCP connections that have completed the 3-way handshake.

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

`max-src-conn-rate` *`number / interval`*
: Limit the rate of new connections to a certain amount per time interval.

Both of these options automatically invoke the `source-track rule` option and are incompatible with `source-track global`.

Since these limits are only being placed on TCP connections that have completed the 3-way handshake, more aggressive actions can be taken on offending IP addresses.

`overload <`*`table`*`>`
: Put an offending host's IP address into the named table.

`flush [global]`
: Kill any other states that match this rule and that were created by this source IP. When `global` is specified, kill all states matching this source IP, regardless of which rule created the state.

An example:

```
table <abusive_hosts> persist
block in quick from <abusive_hosts>

pass in on $ext_if proto tcp to $web_server \
    port www flags S/SA keep state \
    (max-src-conn 100, max-src-conn-rate 15/5, overload
<abusive_hosts> flush)
```

This does the following:

- Limits the maximum number of connections per source to 100
- Rate limits the number of connections to 15 in a 5 second span
- Puts the IP address of any host that breaks these limits into the `<abusive_hosts>` table
- For any offending IP addresses, flush any states created by this rule.

TCP Flags

Matching TCP packets based on flags is most often used to filter TCP packets that are attempting to open a new connection. The TCP flags and their meanings are listed here:

- **F** : FIN **-** Finish; end of session
- **S** : SYN **-** Synchronize; indicates request to start session
- **R** : RST **-** Reset; drop a connection
- **P** : PUSH **-** Push; packet is sent immediately
- **A** : ACK **-** Acknowledgement
- **U** : URG **-** Urgent

- **E** : ECE - Explicit Congestion Notification Echo
- **W** : CWR - Congestion Window Reduced

To have PF inspect the TCP flags during evaluation of a rule, the `flags` keyword is used with the following syntax:

```
     flags check/mask
     flags any
```

The *`mask`* part tells PF to only inspect the specified flags and the *`check`* part specifies which flag(s) must be "on" in the header for a match to occur. Using the `any` keyword allows any combination of flags to be set in the header.

```
     pass in on fxp0 proto tcp from any to any port ssh flags S/SA pass
     in on fxp0 proto tcp from any to any port ssh
```

As `flags S/SA` is set by default, the above rules are equivalent, Each of these rules passes TCP traffic with the SYN flag set while only looking at the SYN and ACK flags. A packet with the SYN and ECE flags would match the above rules, while a packet with SYN and ACK or just ACK would not.

The default flags can be overridden by using the `flags` option as outlined above.

One should be careful with using flags -- understand what you are doing and why, and be careful with the advice people give as a lot of it is bad. Some people have suggested creating state "only if the SYN flag is set and no others". Such a rule would end with:

```
      . . . flags S/FSRPAUEW  bad idea!!
```

The theory is, create state only on the start of the TCP session, and the session should start with a SYN flag, and no others. The problem is some sites are starting to use the ECN flag and any site using ECN that tries to connect to you would be rejected by such a rule. A much better guideline is to not specify any flags at all and let PF apply the default flags to your rules. If you truly need to specify flags yourself then this combination should be safe:

```
     . . . flags S/SAFR
```

While this is practical and safe, it is also unnecessary to check the FIN and RST flags if traffic is also being scrubbed. The scrubbing process will cause PF to drop any incoming packets with illegal TCP flag combinations (such as SYN and RST) and to normalize potentially ambiguous combinations (such as SYN and FIN).

TCP SYN Proxy

Normally when a client initiates a TCP connection to a server, PF will pass the handshake packets between the two endpoints as they arrive. PF has the ability, however, to proxy the handshake. With the handshake proxied, PF itself will complete the handshake with the client, initiate a handshake with the server, and then pass packets between the two. The benefit of this process is that no packets are sent to the server before the client completes the handshake. This eliminates the threat of spoofed TCP SYN floods affecting the server because a spoofed client connection will be unable to complete the handshake.

The TCP SYN proxy is enabled using the `synproxy state` keywords in filter rules. Example:

```
pass in on $ext_if proto tcp to $web_server port www synproxy
state
```

Here, connections to the web server will be TCP proxied by PF.

Because of the way `synproxy state` works, it also includes the same functionality as `keep state` and `modulate state`.

The SYN proxy will not work if PF is running on a bridge(4).

Blocking Spoofed Packets

Address "spoofing" is when an malicious user fakes the source IP address in packets they transmit in order to either hide their real address or to impersonate another node on the network. Once the user has spoofed their address they can launch a network attack without revealing the true source of the attack or attempt to gain access to network services that are restricted to certain IP addresses.

PF offers some protection against address spoofing through the `antispoof` keyword:

```
antispoof [log] [quick] for interface [af]
```

`log`
: Specifies that matching packets should be logged via pflogd(8).

`quick`
: If a packet matches this rule then it will be considered the "winning" rule and ruleset evaluation will stop.

`interface`
: The network interface to activate spoofing protection on. This can also be a list of interfaces.

`af`
: The address family to activate spoofing protection for, either `inet` for IPv4 or `inet6` for IPv6.

Example:

```
antispoof for fxp0 inet
```

When a ruleset is loaded, any occurrences of the `antispoof` keyword are expanded into two filter rules. Assuming that interface `fxp0` has IP address 10.0.0.1 and a subnet mask of 255.255.255.0 (i.e., a /24), the above `antispoof` rule would expand to:

```
block in on ! fxp0 inet from 10.0.0.0/24 to any
block in inet from 10.0.0.1 to any
```

These rules accomplish two things:

- Blocks all traffic coming from the 10.0.0.0/24 network that does *not* pass in through `fxp0`. Since the 10.0.0.0/24 network is on the `fxp0` interface, packets with a source address in that network block should never be seen coming in on any other interface.
- Blocks all incoming traffic from 10.0.0.1, the IP address on `fxp0`. The host machine should never send packets to itself through an external interface, so any incoming packets with a source address belonging to the machine can be considered malicious.

NOTE: The filter rules that the `antispoof` rule expands to will also block packets sent over the loopback interface to local addresses. It's best practice to skip filtering on loopback interfaces anyways, but this becomes a necessity when using antispoof rules:

```
set skip on lo0

antispoof for fxp0 inet
```

Usage of `antispoof` should be restricted to interfaces that have been assigned an IP address. Using `antispoof` on an interface without an IP address will result in filter rules such as:

```
block drop in on ! fxp0 inet all
block drop in inet all
```

With these rules there is a risk of blocking *all* inbound traffic on *all* interfaces.

Unicast Reverse Path Forwarding

PF offers a Unicast Reverse Path Forwarding (uRPF) feature. When a packet is run through the uRPF check, the source IP address of the packet is looked up in the routing table. If the outbound interface found in the routing table entry is the same as the interface that the packet just came in on, then the uRPF check passes. If the interfaces don't match, then it's possible the packet has had its source address spoofed.

The uRPF check can be performed on packets by using the `urpf-failed` keyword in filter rules:

```
block in quick from urpf-failed label uRPF
```

Note that the uRPF check only makes sense in an environment where routing is symmetric.

uRPF provides the same functionality as antispoof rules.

Passive Operating System Fingerprinting

Passive OS Fingerprinting (OSFP) is a method for passively detecting the operating system of a remote host based on certain characteristics within that host's TCP SYN packets. This information can then be used as criteria within filter rules.

PF determines the remote operating system by comparing characteristics of a TCP SYN packet against the fingerprints file, which by default is `/etc/pf.os`. Once PF is enabled, the current fingerprint list can be viewed with this command:

```
# pfctl -s osfp
```

Within a filter rule, a fingerprint may be specified by OS class, version, or subtype/patch level. Each of these items is listed in the output of the `pfctl` command shown above. To specify a fingerprint in a filter rule, the `os` keyword is used:

```
pass  in on $ext_if from any os OpenBSD keep state
block in on $ext_if from any os "Windows 2000"
block in on $ext_if from any os "Linux 2.4 ts"
block in on $ext_if from any os unknown
```

The special operating system class `unknown` allows for matching packets when the OS fingerprint is not known.

TAKE NOTE of the following:

- Operating system fingerprints are occasionally wrong due to spoofed and/or crafted packets that are made to look like they originated from a specific operating system.
- Certain revisions or patchlevels of an operating system may change the stack's behavior and cause it to either not match what's in the fingerprints file or to match another entry altogether.
- OSFP only works on the TCP SYN packet; it will not work on other protocols or on already established connections.

IP Options

By default, PF blocks packets with IP options set. This can make the job more difficult for "OS fingerprinting" utilities like nmap. If you have an application that requires the passing of these packets, such as multicast or IGMP, you can use the `allow-opts` directive:

```
pass in quick on fxp0 all allow-opts
```

Filtering Ruleset Example

Below is an example of a filtering ruleset. The machine running PF is acting as a firewall between a small, internal network and the Internet. Only the filter rules are shown; queueing, `nat`, `rdr`, etc., have been left out of this example.

```
ext_if  = "fxp0"
int_if  = "dc0"
lan_net = "192.168.0.0/24"

# table containing all IP addresses assigned to the firewall
table <firewall> const { self }

# don't filter on the loopback interface
set skip on lo0

# scrub incoming packets
match in all scrub (no-df)

# setup a default deny policy
block all

# activate spoofing protection for all interfaces
block in quick from urpf-failed

# only allow ssh connections from the local network if it's from the
# trusted computer, 192.168.0.15. use "block return" so that a TCP RST is
# sent to close blocked connections right away. use "quick" so that this
# rule is not overridden by the "pass" rules below.
block return in quick on $int_if proto tcp from ! 192.168.0.15 \
   to $int_if port ssh

# pass all traffic to and from the local network.
# these rules will create state entries due to the default
# "keep state" option which will automatically be applied.
pass in  on $int_if from $lan_net
pass out on $int_if to $lan_net

# pass tcp, udp, and icmp out on the external (Internet) interface.
# tcp connections will be modulated, udp/icmp will be tracked
# statefully.
pass out on $ext_if proto { tcp udp icmp } all modulate state

# allow ssh connections in on the external interface as long as they're
# NOT destined for the firewall (i.e., they're destined for a machine on
# the local network). log the initial packet so that we can later tell
# who is trying to connect. use the tcp syn proxy to proxy the connection.
# the default flags "S/SA" will automatically be applied to the rule by
# PF.
pass in log on $ext_if proto tcp to ! <firewall> \
   port ssh synproxy state
```

[Previous: Tables] [Contents] [Next: Network Address Translation]

---

www@openbsd.org

$OpenBSD: filter.html,v 1.54 2010/05/19 13:25:16 sthen Exp $

[Previous: Packet Filtering] [Contents] [Next: Traffic Redirection (Port Forwarding)]

PF: Network Address Translation (NAT)

Table of Contents

Introduction

Network Address Translation (NAT) is a way to map an entire network (or networks) to a single IP address. NAT is necessary when the number of IP addresses assigned to you by your Internet Service Provider is less than the total number of computers that you wish to provide Internet access for. NAT is described in RFC 1631, "The IP Network Address Translator (NAT)."

NAT allows you to take advantage of the reserved address blocks described in RFC 1918, "Address Allocation for Private Internets." Typically, your internal network will be setup to use one or more of these network blocks. They are:

```
        10.0.0.0/8       (10.0.0.0 - 10.255.255.255)
        172.16.0.0/12    (172.16.0.0 - 172.31.255.255)
        192.168.0.0/16   (192.168.0.0 - 192.168.255.255)
```

An OpenBSD system doing NAT will have at least two network adapters, one to the Internet, the other to your internal network. NAT will be translating requests from the internal network so they appear to all be coming from your OpenBSD NAT system.

How NAT Works

When a client on the internal network contacts a machine on the Internet, it sends out IP packets destined for that machine. These packets contain all the addressing information necessary to get them to their destination. NAT is concerned with these pieces of information:

- Source IP address (for example, 192.168.1.35)
- Source TCP or UDP port (for example, 2132)

When the packets pass through the NAT gateway they will be modified so that they appear to be coming from the NAT gateway itself. The NAT gateway will record the changes it makes in its state table so that it can a) reverse the changes on return packets and b) ensure that return packets are passed through the firewall and are not blocked. For example, the following changes might be made:

- Source IP: replaced with the external address of the gateway (for example, 24.5.0.5)
- Source port: replaced with a randomly chosen, unused port on the gateway (for example, 53136)

Neither the internal machine nor the Internet host is aware of these translation steps. To the internal machine, the NAT system is simply an Internet gateway. To the Internet host, the packets appear to come directly from the NAT system; it is completely unaware that the internal workstation even exists.

When the Internet host replies to the internal machine's packets, they will be addressed to the NAT gateway's external IP (24.5.0.5) at the translation port (53136). The NAT gateway will then search the state table to determine if the reply packets match an already established connection. A unique match will be found based on the IP/port combination which tells PF the packets belong to a connection initiated by the internal machine 192.168.1.35. PF will then make the opposite changes it made to the outgoing packets and forward the reply packets on to the internal machine.

Translation of ICMP packets happens in a similar fashion but without the source port modification.

IP Forwarding

Since NAT is almost always used on routers and network gateways, it will probably be necessary to enable IP forwarding so that packets can travel between network interfaces on the OpenBSD machine. IP forwarding is enabled using the sysctl(3) mechanism:

```
# sysctl net.inet.ip.forwarding=1
# sysctl net.inet6.ip6.forwarding=1 (if using IPv6)
```

To make this change permanent, the following lines should be added to `/etc/sysctl.conf`:

```
net.inet.ip.forwarding=1
net.inet6.ip6.forwarding=1
```

These lines are present but commented out (prefixed with a #) in the default install. Remove the # and save the file. IP forwarding will be enabled when the machine is rebooted.

Configuring NAT

NOTE: This information is for OpenBSD 4.7. NAT configuration was significantly different in earlier versions.

NAT is specified as an optional `nat-to` parameter to an outbound `pass` rule. Often, rather than being set directly on the `pass` rule, a `match` rule is used. When a packet is selected by a `match` rule, parameters (e.g. `nat-to`) in that rule are remembered and are applied to the packet when a `pass` rule matching the packet is reached. This permits a whole class of packets to be handled by a single `match` rule and then specific decisions on whether to allow the traffic can be made with `block` and `pass` rules.

The general format in `pf.conf` looks something like this:

```
match out on interface [af] \
   from src_addr to dst_addr \
   nat-to ext_addr [pool_type] [static-port]
...
pass out [log] on interface [af] [proto protocol] \
   from ext_addr [port src_port] \
   to dst_addr [port dst_port]
```

`match`

> When a packet traverses the ruleset and matches a `match` rule, any optional parameters specified in that rule are remembered for future use (made "sticky").

`pass`

> This rule allows the packet to be transmitted. If the packet was previously matched by a `match` rule where parameters were specified, they will be applied to this packet. `pass` rules may have their own parameters; these take priority over parameters specified in a `match` rule.

`out`

> Specifies the direction of packet flow where this rule applies. `nat-to` may only be specified for outbound packets.

`log`

> Log matching packets via pflogd(8). Normally only the first packet that matches will be logged. To log all matching packets, use `log (all)`.

interface
The name or group of the network interface to transmit packets on.

af
The address family, either `inet` for IPv4 or `inet6` for IPv6. PF is usually able to determine this parameter based on the source/destination address(es).

protocol
The protocol (e.g. tcp, udp, icmp) of packets to allow. If *src_port* or *dst_port* is specified, the protocol *must* also be given.

src_addr
The source (internal) address of packets that will be translated. The source address can be specified as:

- A single IPv4 or IPv6 address.
- A CIDR network block.
- A fully qualified domain name that will be resolved via DNS when the ruleset is loaded. All resulting IP addresses will be substituted into the rule.
- The name or group of a network interface. Any IP addresses assigned to the interface will be substituted into the rule at load time.
- The name of a network interface followed by */netmask* (e.g. `/24`). Each IP address on the interface is combined with the netmask to form a CIDR network block which is substituted into the rule.
- The name or group of a network interface followed by any one of these modifiers:
  - `:network` - substitutes the CIDR network block (e.g., 192.168.0.0/24)
  - `:broadcast` - substitutes the network broadcast address (e.g., 192.168.0.255)
  - `:peer` - substitutes the peer's IP address on a point-to-point link

  In addition, the `:0` modifier can be appended to either an interface name/group or to any of the above modifiers to indicate that PF should not include aliased IP addresses in the substitution. These modifiers can also be used when the interface is contained in parentheses. Example: `fxp0:network:0`
- A table.
- Any of the above but negated using the `!` ("not") modifier.
- A set of addresses using a list.
- The keyword `any` meaning all addresses

src_port
The source port in the Layer 4 packet header. Ports can be specified as:

- A number between 1 and 65535
- A valid service name from `/etc/services`
- A set of ports using a list
- A range:
  - `!=` (not equal)
  - `<` (less than)
  - `>` (greater than)
  - `<=` (less than or equal)

  - `>=` (greater than or equal)
  - `><` (range)
  - `<>` (inverse range)

    The last two are binary operators (they take two arguments) and do not include the arguments in the range.
  - `:` (inclusive range)

    The inclusive range operator is also a binary operator and does include the arguments in the range.

The `port` option is not usually used in `nat` rules because the goal is usually to NAT all traffic regardless of the port(s) being used.

`dst_addr`

The destination address of packets to be translated. The destination address is specified in the same way as the source address.

`dst_port`

The destination port in the Layer 4 packet header. This port is specified in the same way as the source port.

`ext_addr`

The external (translation) address on the NAT gateway that packets will be translated to. The external address can be specified as:

- A single IPv4 or IPv6 address.
- A CIDR network block.
- A fully qualified domain name that will be resolved via DNS when the ruleset is loaded.
- The name of the external network interface. Any IP addresses assigned to the interface will be substituted into the rule at load time.
- The name of the external network interface in parentheses `( )`. This tells PF to update the rule if the IP address(es) on the named interface changes. This is highly useful when the external interface gets its IP address via DHCP or dial-up as the ruleset doesn't have to be reloaded each time the address changes.
- The name of a network interface followed by either one of these modifiers:
  - `:network` - substitutes the CIDR network block (e.g., 192.168.0.0/24)
  - `:peer` - substitutes the peer's IP address on a point-to-point link

  In addition, the `:0` modifier can be appended to either an interface name or to any of the above modifiers to indicate that PF should not include aliased IP addresses in the substitution. These modifiers can also be used when the interface is contained in parentheses. Example: `fxp0:network:0`
- A set of addresses using a list.

`pool_type`

Specifies the type of address pool to use for translation.

`static-port`

Tells PF not to translate the source port in TCP and UDP packets.

This would lead to a most basic form of these lines similar to this:

```
     match out on tl0 from 192.168.1.0/24 to any nat-to 24.5.0.5
     pass on tl0 from 192.168.1.0/24 to any
```

or you may simply use

```
     pass out on tl0 from 192.168.1.0/24 to any nat-to 24.5.0.5
```

This rule says to perform NAT on the `tl0` interface for any packets coming from 192.168.1.0/24 and to replace the source IP address with 24.5.0.5.

While the above rule is correct, it is not recommended form. Maintenance could be difficult as any change of the external or internal network numbers would require the line be changed. Compare instead with this easier to maintain line (`tl0` is external, `dc0` internal):

```
     pass out on tl0 from dc0:network to any nat-to tl0
```

The advantage should be fairly clear: you can change the IP addresses of either interface without changing this rule.

When specifying an interface name for the translation address as above, the IP address is determined at pf.conf *load* time, not on the fly. If you are using DHCP to configure your external interface, this can be a problem. If your assigned IP address changes, NAT will continue translating outgoing packets using the old IP address. This will cause outgoing connections to stop functioning. To get around this, you can tell PF to automatically update the translation address by putting parentheses around the interface name:

```
     pass out on tl0 from dc0:network to any nat-to (tl0)
```

This method works for translation to both IPv4 and IPv6 addresses.

Bidirectional Mapping (1:1 mapping)

A bidirectional mapping can be established by using the `binat-to` parameter. A `binat-to` rule establishes a one to one mapping between an internal IP address and an external address. This can be useful, for example, to provide a web server on the internal network with its own external IP address. Connections from the Internet to the external address will be translated to the internal address and connections from the web server (such as DNS requests) will be translated to the external address. TCP and UDP ports are never modified with `binat-to` rules as they are with `nat` rules.

Example:

```
     web_serv_int = "192.168.1.100"
     web_serv_ext = "24.5.0.6"

     pass on tl0 from $web_serv_int to any binat-to $web_serv_ext
```

Translation Rule Exceptions

If you need to translate most traffic, but provide exceptions in some cases, make sure that the exceptions are handled by a filter rule which does not include the nat-to parameter. For example, if the NAT example above was modified to look like this:

```
     pass out on tl0 from 192.168.1.0/24 to any nat-to
     24.2.74.79 pass out on tl0 from 192.168.1.208 to any
```

Then the entire 192.168.1.0/24 network would have its packets translated to the external address 24.2.74.79 except for 192.168.1.208.

Checking NAT Status

To view the active NAT translations pfctl(8) is used with the `-s state` option. This option will list all the current NAT sessions:

```
   # pfctl -s state
   fxp0 TCP 192.168.1.35:2132 -> 24.5.0.5:53136 -> 65.42.33.245:22
TIME_WAIT:TIME_WAIT
   fxp0 UDP 192.168.1.35:2491 -> 24.5.0.5:60527 -> 24.2.68.33:53
MULTIPLE:SINGLE
```

Explanations (first line only):

fxp0
: Indicates the interface that the state is bound to. The word `self` will appear if the state is `floating`.

TCP
: The protocol being used by the connection.

192.168.1.35:2132
: The IP address (192.168.1.35) of the machine on the internal network. The source port (2132) is shown after the address. This is also the address that is replaced in the IP header.

24.5.0.5:53136

The IP address (24.5.0.5) and port (53136) on the gateway that packets are being translated to.

65.42.33.245:22

The IP address (65.42.33.245) and the port (22) that the internal machine is connecting to.

TIME_WAIT:TIME_WAIT

This indicates what state PF believes the TCP connection to be in.

[Previous: Packet Filtering] [Contents] [Next: Traffic Redirection (Port Forwarding)]

www@openbsd.org

$OpenBSD: nat.html,v 1.32 2010/05/19 13:25:16 sthen Exp $

[Previous: Network Address Translation] [Contents] [Next: Shortcuts For Creating Rulesets]

PF: Redirection (Port Forwarding)

Table of Contents

Introduction

When you have NAT running in your office you have the entire Internet available to all your machines. What if you have a machine behind the NAT gateway that needs to be accessed from outside? This is where redirection comes in. Redirection allows incoming traffic to be sent to a machine behind the NAT gateway.

Let's look at an example:

```
pass in on tl0 proto tcp from any to any port 80 rdr-to
192.168.1.20
```

This line redirects TCP port 80 (web server) traffic to a machine inside the network at 192.168.1.20. So, even though 192.168.1.20 is behind your gateway and inside your network, the outside world can access it.

The `from any to any` part of the above `rdr` line can be quite useful. If you know what addresses or subnets are supposed to have access to the web server at port 80, you can restrict them here:

```
     pass in on tl0 proto tcp from 27.146.49.0/24 to any port 80
     \
        rdr-to 192.168.1.20
```

This will redirect only the specified subnet. Note this implies you can redirect different incoming hosts to different machines behind the gateway. This can be quite useful. For example, you could have users at remote sites access their own desktop computers using the same port and IP address on the gateway as long as you know the IP address they will be connecting from:

```
     pass in on tl0 proto tcp from 27.146.49.14 to any port 80 \
        rdr-to 192.168.1.20
     pass in on tl0 proto tcp from 16.114.4.89 to any port 80 \
        rdr-to 192.168.1.22
     pass in on tl0 proto tcp from 24.2.74.178 to any port 80 \
        rdr-to 192.168.1.23
```

A range of ports can also be redirected within the same rule:

```
     pass in on tl0 proto tcp from any to any port 5000:5500 \
        rdr-to 192.168.1.20
     pass in on tl0 proto tcp from any to any port 5000:5500 \
        rdr-to 192.168.1.20 port 6000
     pass in on tl0 proto tcp from any to any port 5000:5500 \
        rdr-to 192.168.1.20 port 7000:*
```

These examples show ports 5000 to 5500 inclusive being redirected to 192.168.1.20. In rule #1, port 5000 is redirected to 5000, 5001 to 5001, etc. In rule #2, the entire port range is redirected to port 6000. And in rule #3, port 5000 is redirected to 7000, 5001 to 7001, etc.

Security Implications

Redirection does have security implications. Punching a hole in the firewall to allow traffic into the internal, protected network potentially opens up the internal machine to compromise. If traffic is forwarded to an internal web server for example, and a vulnerability is discovered in the web server daemon or in a CGI script run on the web server, then that machine can be compromised from an intruder on the Internet. From there, the intruder has a doorway to the internal network, one that is permitted to pass right through the firewall.

These risks can be minimized by keeping the externally accessed system tightly confined on a separate network. This network is often referred to as a Demilitarized Zone (DMZ) or a Private Service Network

(PSN). This way, if the web server is compromised, the effects can be limited to the DMZ/PSN network by careful filtering of the traffic permitted to and from the DMZ/PSN.

Redirection and Reflection

Often, redirection rules are used to forward incoming connections from the Internet to a local server with a private address in the internal network or LAN, as in:

```
server = 192.168.1.40

pass in on $ext_if proto tcp from any to $ext_if port 80 \
   rdr-to $server port 80
```

But when the redirection rule is tested from a client on the LAN, it doesn't work. The reason is that redirection rules apply only to packets that pass through the specified interface (`$ext_if`, the external interface, in the example). Connecting to the external address of the firewall from a host on the LAN, however, does not mean the packets will actually pass through its external interface. The TCP/IP stack on the firewall compares the destination address of incoming packets with its own addresses and aliases and detects connections to itself as soon as they have passed the internal interface. Such packets do not physically pass through the external interface, and the stack does not simulate such a passage in any way. Thus, PF never sees these packets on the external interface, and the redirection rule, specifying the external interface, does not apply.

Adding a second redirection rule for the internal interface does not have the desired effect either. When the local client connects to the external address of the firewall, the initial packet of the TCP handshake reaches the firewall through the internal interface. The redirection rule does apply and the destination address gets replaced with that of the internal server. The packet gets forwarded back through the internal interface and reaches the internal server. But the source address has not been translated, and still contains the local client's address, so the server sends its replies directly to the client. The firewall never sees the reply and has no chance to properly reverse the translation. The client receives a reply from a source it never expected and drops it. The TCP handshake then fails and no connection can be established.

Still, it's often desirable for clients on the LAN to connect to the same internal server as external clients and to do so transparently. There are several solutions for this problem:

Split-Horizon DNS

It's possible to configure DNS servers to answer queries from local hosts differently than external queries so that local clients will receive the internal server's address during name resolution. They will then connect directly to the local server, and the firewall isn't involved at all. This reduces local traffic since packets don't have to be sent through the firewall.

Moving the Server Into a Separate Local Network

Adding an additional network interface to the firewall and moving the local server from the client's network into a dedicated network (DMZ) allows redirecting of connections from local clients in the same way as the redirection of external connections. Use of separate networks has several advantages, including improving security by isolating the server from the remaining local hosts. Should the server (which in our case is reachable from the Internet) ever become compromised, it can't access other local hosts directly as all connections have to pass through the firewall.

TCP Proxying

A generic TCP proxy can be setup on the firewall, either listening on the port to be forwarded or getting connections on the internal interface redirected to the port it's listening on. When a local client connects to the firewall, the proxy accepts the connection, establishes a second connection to the internal server, and forwards data between those two connections.

Simple proxies can be created using inetd(8) and nc(1). The following `/etc/inetd.conf` entry creates a listening socket bound to the loopback address (127.0.0.1) and port 5000. Connections are forwarded to port 80 on server 192.168.1.10. The forwarding is done by user "proxy".

```
127.0.0.1:5000 stream tcp nowait proxy /usr/bin/nc nc -w \
   20 192.168.1.10 80
```

The following redirection rule forwards port 80 on the internal interface to the proxy:

```
pass in on $int_if proto tcp from $int_net to $ext_if port
80 \
   rdr-to 127.0.0.1 port 5000
```

High-performance proxies may also be created with relayd(8).

RDR-TO and NAT-TO Combination

With an additional NAT rule on the internal interface, the lacking source address translation described above can be achieved.

```
pass in on $int_if proto tcp from $int_net to $ext_if port
80 \
   rdr-to $server
pass out on $int_if proto tcp to $server port 80 \
        received-on $int_if nat-to $int_if
```

This will cause the initial packet from the client to be translated again when it's forwarded back through the internal interface, replacing the client's source address with the firewall's internal address. The internal server will reply back to the firewall, which can reverse both NAT and RDR translations when forwarding to the local client. This construct is rather complex as it creates two separate states for each reflected connection. Care must be taken to prevent the NAT rule from applying to other traffic, for instance connections originating from external hosts (through other redirections) or the firewall itself. Note that the `rdr-to` rule above will cause the TCP/IP stack to see packets arriving on the internal interface with a destination address inside the internal network.

In general, the previously mentioned solutions should be used instead.

[Previous: Network Address Translation] [Contents] [Next: Shortcuts For Creating Rulesets]

---

www@openbsd.org

$OpenBSD: rdr.html,v 1.30 2010/05/24 12:13:15 sthen Exp $

[Previous: Traffic Redirection (Port Forwarding)] [Contents] [Next: Runtime Options]

PF: Shortcuts For Creating Rulesets

Table of Contents

Introduction

PF offers many ways in which a ruleset can be simplified. Some good examples are by using macros and lists. In addition, the ruleset language, or grammar, also offers some shortcuts for making a ruleset simpler. As a general rule of thumb, the simpler a ruleset is, the easier it is to understand and to maintain.

Using Macros

Macros are useful because they provide an alternative to hard-coding addresses, port numbers, interfaces names, etc., into a ruleset. Did a server's IP address change? No problem, just update the macro; no need to mess around with the filter rules that you've spent time and energy perfecting for your needs.

A common convention in PF rulesets is to define a macro for each network interface. If a network card ever needs to be replaced with one that uses a different driver, for example swapping out a 3Com for an Intel, the macro can be updated and the filter rules will function as before. Another benefit is when installing the same ruleset on multiple machines. Certain machines may have different network cards in them, and using macros to define the network interfaces allows the rulesets to be installed with minimal editing. Using macros to define information in a ruleset that is subject to change, such as port numbers, IP addresses, and interface names, is recommended practice.

```
# define macros for each network interface
IntIF = "dc0"
ExtIF = "fxp0"
DmzIF = "fxp1"
```

Another common convention is using macros to define IP addresses and network blocks. This can greatly reduce the maintenance of a ruleset when IP addresses change.

```
# define our networks
IntNet = "192.168.0.0/24"
ExtAdd = "24.65.13.4"
DmzNet = "10.0.0.0/24"
```

If the internal network ever expanded or was renumbered into a different IP block, the macro can be updated:

```
IntNet = "{ 192.168.0.0/24, 192.168.1.0/24 }"
```

Once the ruleset is reloaded, everything will work as before.

Using Lists

Let's look at a good set of rules to have in your ruleset to handle RFC 1918 addresses that just shouldn't be floating around the Internet, and when they are, are usually trying to cause trouble:

```
block in  quick on tl0 inet from 127.0.0.0/8 to any
block in  quick on tl0 inet from 192.168.0.0/16 to any
block in  quick on tl0 inet from 172.16.0.0/12 to any
block in  quick on tl0 inet from 10.0.0.0/8 to any
block out quick on tl0 inet from any to 127.0.0.0/8
block out quick on tl0 inet from any to 192.168.0.0/16
block out quick on tl0 inet from any to 172.16.0.0/12
block out quick on tl0 inet from any to 10.0.0.0/8
```

Now look at the following simplification:

```
block in  quick on tl0 inet from { 127.0.0.0/8,
192.168.0.0/16, \
   172.16.0.0/12, 10.0.0.0/8 } to any
block out quick on tl0 inet from any to { 127.0.0.0/8, \
   192.168.0.0/16, 172.16.0.0/12, 10.0.0.0/8 }
```

The ruleset has been reduced from eight lines down to two. Things get even better when macros are used in conjunction with a list:

```
NoRouteIPs = "{ 127.0.0.0/8, 192.168.0.0/16, 172.16.0.0/12,
\
   10.0.0.0/8 }"
ExtIF = "tl0"
block in  quick on $ExtIF from $NoRouteIPs to any
block out quick on $ExtIF from any to $NoRouteIPs
```

Note that macros and lists simplify the `pf.conf` file, but the lines are actually expanded by pfctl(8) into multiple rules. So, the above example actually expands to the following rules:

```
block in  quick on tl0 inet from 127.0.0.0/8 to any
block in  quick on tl0 inet from 192.168.0.0/16 to any
block in  quick on tl0 inet from 172.16.0.0/12 to any
block in  quick on tl0 inet from 10.0.0.0/8 to any
block out quick on tl0 inet from any to 10.0.0.0/8
block out quick on tl0 inet from any to 172.16.0.0/12
block out quick on tl0 inet from any to 192.168.0.0/16
block out quick on tl0 inet from any to 127.0.0.0/8
```

As you can see, the PF expansion is purely a convenience for the writer and maintainer of the `pf.conf` file, not an actual simplification of the rules processed by pf(4).

Macros can be used to define more than just addresses and ports; they can be used anywhere in a PF rules file:

```
pre = "pass in quick on ep0 inet proto tcp from "
post = "to any port { 80, 6667 }"

# David's classroom
$pre 21.14.24.80 $post

# Nick's home
$pre 24.2.74.79 $post
$pre 24.2.74.178 $post
```

Expands to:

```
    pass in quick on ep0 inet proto tcp from 21.14.24.80 to any
    \
       port = 80
    pass in quick on ep0 inet proto tcp from 21.14.24.80 to any
    \
       port = 6667
    pass in quick on ep0 inet proto tcp from 24.2.74.79 to any \
       port = 80
    pass in quick on ep0 inet proto tcp from 24.2.74.79 to any \
       port = 6667
    pass in quick on ep0 inet proto tcp from 24.2.74.178 to any
    \
       port = 80
    pass in quick on ep0 inet proto tcp from 24.2.74.178 to any
    \
       port = 6667
```

PF Grammar

Packet Filter's grammar is quite flexible which, in turn, allows for great flexibility in a ruleset. PF is able to infer certain keywords which means that they don't have to be explicitly stated in a rule, and keyword ordering is relaxed such that it isn't necessary to memorize strict syntax.

Elimination of Keywords

To define a "default deny" policy, two rules are used:

```
    block in  all
    block out all
```

This can now be reduced to:

```
    block
```

When no direction is specified, PF will assume the rule applies to packets moving in both directions.

Similarly, the "`from any to any`" and "`all`" clauses can be left out of a rule, for example:

```
    block in on rl0 all
```

```
     pass  in quick log on rl0 proto tcp from any to any port 22
     keep state
```

can be simplified as:

```
     block in on rl0
     pass  in quick log on rl0 proto tcp to port 22 keep state
```

The first rule blocks all incoming packets from anywhere to anywhere on rl0, and the second rule passes in TCP traffic on rl0 to port 22.

Return Simplification

A ruleset used to block packets and reply with a TCP RST or ICMP Unreachable response could look like this:

```
     block in all
     block return-rst in proto tcp all
     block return-icmp in proto udp all
     block out all
     block return-rst out proto tcp all
     block return-icmp out proto udp all
```

This can be simplified as:

```
     block return
```

When PF sees the `return` keyword, it's smart enough to send the proper response, or no response at all, depending on the protocol of the packet being blocked.

Keyword Ordering

The order in which keywords are specified is flexible in most cases. For example, a rule written as:

```
     pass in log quick on rl0 proto tcp to port 22 \
        flags S/SA keep state queue ssh label ssh
```

Can also be written as:

```
     pass in quick log on rl0 proto tcp to port 22 \
        queue ssh keep state label ssh flags S/SA
```

Other, similar variations will also work.

[Previous: Traffic Redirection (Port Forwarding)] [Contents] [Next: Runtime Options]

---

www@openbsd.org

$OpenBSD: shortcuts.html,v 1.25 2010/05/19 13:25:16 sthen Exp $

OpenBSD

[Previous: Shortcuts For Creating Rulesets] [Contents] [Next: Anchors]

PF: Runtime Options

Options are used to control PF's operation. Options are specified in `pf.conf` using the `set` directive.

`set block-policy` *`option`*

Sets the default behavior for filter rules that specify the `block` action.

- `drop` - packet is silently dropped.
- `return` - a TCP RST packet is returned for blocked TCP packets and an ICMP Unreachable packet is returned for all others.

Note that individual filter rules can override the default response. The default is `drop`.

`set debug` *`option`*

Set pf's debugging level.

- `none` - no debugging messages are shown.
- `urgent` - debug messages generated for serious errors.
- `misc` - debug messages generated for various errors (e.g., to see status from the packet normalizer/scrubber and for state creation failures).
- `loud` - debug messages generated for common conditions (e.g., to see status from the passive OS fingerprinter).

The default is `urgent`.

`set fingerprints` *`file`*

Sets the file to load operating system fingerprints from. For use with passive OS fingerprinting. The default is `/etc/pf.os`.

`set limit` *`option value`*

Set various limits on pf's operation.

- `frags` - maximum number of entries in the memory pool used for packet reassembly (scrub rules). Default is 5000.
- `src-nodes` - maximum number of entries in the memory pool used for tracking source IP addresses (generated by the `sticky-address` and `source-track` options). Default is 10000.
- `states` - maximum number of entries in the memory pool used for state table entries (filter rules that specify `keep state`). Default is 10000.
- `tables` - maximum number of tables that can be created. The default is 1000.
- `table-entries` - the overall limit on how many addresses can be stored in all tables. The default is 200000. If the system has less than 100MB of physical memory, the default is set to 100000.

`set loginterface` *`interface`*

Sets the interface for which PF should gather statistics such as bytes in/out and packets passed/blocked. Statistics can only be gathered for *one* interface at a time. Note that the `match`, `bad-offset`, etc., counters and the state table counters are recorded regardless of whether `loginterface` is set or not. To turn this option off, set it to `none`. The default is `none`.

`set optimization` *`option`*

Optimize PF for one of the following network environments:

- `normal` - suitable for almost all networks.
- `high-latency` - high latency networks such as satellite connections.
- `aggressive` - aggressively expires connections from the state table. This can greatly reduce the memory requirements on a busy firewall at the risk of dropping idle connections early.
- `conservative` - extremely conservative settings. This avoids dropping idle connections at the expense of greater memory utilization and slightly increased processor utilization.

The default is `normal`.

`set ruleset-optimization` *`option`*

Control operation of the PF ruleset optimizer.

- `none` - disable the optimizer altogether.
- `basic` - enables the following ruleset optimizations:
  1. remove duplicate rules
  2. remove rules that are a subset of another rule
  3. combine multiple rules into a table when advantageous
  4. re-order the rules to improve evaluation performance
- `profile` - uses the currently loaded ruleset as a feedback profile to tailor the ordering of quick rules to actual network traffic.

Starting in OpenBSD 4.2, the default is `basic`. See pf.conf(5) for a more complete description.

`set skip on` *`interface`*

Skip *all* PF processing on *`interface`*. This can be useful on loopback interfaces where filtering, normalization, queueing, etc, are not required. This option can be used multiple times. By default this option is not set.

`set state-policy` *`option`*

Sets PF's behavior when it comes to keeping state. This behavior can be overridden on a per rule basis. See Keeping State.

- `if-bound` - states are bound to the interface they're created on. If traffic matches a state table entry but is not crossing the interface recorded in that state entry, the match is rejected. The packet must then match a filter rule or will be dropped/rejected altogether.
- `floating` - states can match packets on any interface. As long as the packet matches a state entry and is passing in the same direction as it was on the interface when the state was created, it does not matter what interface it's crossing, it will pass.

The default is `floating`.

`set timeout` *`option value`*

Set various timeouts (in seconds).

- `interval` - seconds between purges of expired states and packet fragments. The default is `10`.

  - `frag` - seconds before an unassembled fragment is expired. The default is `30`.
  - `src.track` - seconds to keep a source tracking entry in memory after the last state expires. The default is `0` (zero).

Example:

```
set timeout interval 10
set timeout frag 30
set limit { frags 5000, states 2500 }
set optimization high-latency
set block-policy return
set loginterface dc0
set fingerprints "/etc/pf.os.test"
set skip on lo0
set state-policy if-bound
```

[Previous: Shortcuts For Creating Rulesets] [Contents] [Next: anchors]

---

www@openbsd.org

$OpenBSD: options.html,v 1.23 2010/05/19 13:25:16 sthen Exp $

OpenBSD

[Previous: Runtime Options] [Contents] [Next: Packet Queueing and Prioritization]

PF: Anchors

Table of Contents

Introduction

In addition to the main ruleset, PF can also evaluate sub rulesets. Since sub rulesets can be manipulated on the fly by using pfctl(8), they provide a convenient way of dynamically altering an active ruleset. Whereas a table is used to hold a dynamic list of addresses, a sub ruleset is used to hold a dynamic set of rules. A sub ruleset is attached to the main ruleset by using an `anchor`.

Anchors can be nested which allows for sub rulesets to be chained together. Anchor rules will be evaluated relative to the anchor in which they are loaded. For example, anchor rules in the main ruleset will create anchor attachment points with the main ruleset as their parent, and anchor rules loaded from files with the `load anchor` directive will create anchor points with that anchor as their parent.

Anchors

An anchor is a collection of rules, tables, and other anchors that has been assigned a name. When PF comes across an `anchor` rule in the main ruleset, it will evaluate the rules contained within the anchor point as it evaluates rules in the main ruleset. Processing will then continue in the main ruleset unless the packet matches a filter rule that uses the `quick` option, in which case the match will be considered final and will abort the evaluation of rules in both the anchor and the main rulesets.

For example:

```
ext_if = "fxp0"

block on $ext_if
pass  out on $ext_if
anchor goodguys
```

This ruleset sets a default deny policy on `fxp0` for both incoming and outgoing traffic. Traffic is then statefully passed out and an anchor rule is created named `goodguys`. Anchors can be populated with rules by three methods:

- using a `load` rule
- using pfctl(8)
- specifying the rules inline of the main ruleset

The `load` rule causes `pfctl` to populate the specified anchor by reading rules from a text file. The `load` rule must be placed after the `anchor` rule. Example:

```
anchor goodguys
load anchor goodguys from "/etc/anchor-goodguys-ssh"
```

To add rules to an anchor using `pfctl`, the following type of command can be used:

```
# echo "pass in proto tcp from 192.0.2.3 to any port 22" \
   | pfctl -a goodguys -f -
```

Rules can also be saved and loaded from a text file:

```
# cat >> /etc/anchor-goodguys-www
pass in proto tcp from 192.0.2.3 to any port 80
pass in proto tcp from 192.0.2.4 to any port { 80 443 }

# pfctl -a goodguys -f /etc/anchor-goodguys-www
```

To load rules directly from the main ruleset, enclose the anchor rules in a brace-delimited block:

```
anchor "goodguys" {
   pass in proto tcp from 192.168.2.3 to port 22
}
```

Inline anchors can also contain more anchors.

```
allow = "{ 192.0.2.3 192.0.2.4 }"

anchor "goodguys" {
   anchor {
      pass in proto tcp from 192.0.2.3 to port 80
   }
   pass in proto tcp from $allow to port 22
}
```

With inline anchors the name of the anchor becomes optional. Note how the nested anchor in the above example does not have a name. Also note how the macro `$allow` is created outside of the anchor (in the main ruleset) and is then used within the anchor.

Rules can be loaded into an anchor using the same syntax and options as rules loaded into the main ruleset. One caveat, however, is that unless you're using inline anchors any macros that are used must also be defined within the anchor itself; macros that are defined in the parent ruleset are *not* visible from the anchor.

Since anchors can be nested, it's possible to specify that all child anchors within a specified anchor be evaluated:

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

This syntax causes each rule within each anchor attached to the `spam` anchor to be evaluated. The child anchors will be evaluated in alphabetical order but are not descended into recursively. Anchors are always evaluated relative to the anchor in which they're defined.

Each anchor, as well as the main ruleset, exist separately from the other rulesets. Operations done on one ruleset, such as flushing the rules, do not affect any of the others. In addition, removing an anchor point from the main ruleset does not destroy the anchor or any child anchors that are attached to that anchor. An anchor is not destroyed until it's flushed of all rules using pfctl(8) and there are no child anchors within the anchor.

Anchor Options

Optionally, `anchor` rules can specify interface, protocol, source and destination address, tag, etc., using the same syntax as other rules. When such information is given, `anchor` rules are only processed if the packet matches the `anchor` rule's definition. For example:

```
ext_if = "fxp0"

block on $ext_if
```

```
pass  out on $ext_if
anchor ssh in on $ext_if proto tcp to port 22
```

The rules in the anchor `ssh` are only evaluated for TCP packets destined for port 22 that come in on `fxp0`. Rules are then added to the `anchor` like so:

```
# echo "pass in from 192.0.2.10 to any" | pfctl -a ssh -f -
```

So, even though the filter rule doesn't specify an interface, protocol, or port, the host 192.0.2.10 will only be permitted to connect using SSH because of the `anchor` rule's definition.

The same syntax can be applied to inline anchors.

```
allow = "{ 192.0.2.3 192.0.2.4 }"

anchor "goodguys" in proto tcp {
   anchor proto tcp to port 80 {
      pass from 192.0.2.3
   }
   anchor proto tcp to port 22 {
      pass from $allow
   }
}
```

Manipulating Anchors

Manipulation of anchors is performed via `pfctl`. It can be used to add and remove rules from an anchor without reloading the main ruleset.

To list all the rules in the anchor named `ssh`:

```
# pfctl -a ssh -s rules
```

To flush all rules from the same anchor:

```
# pfctl -a ssh -F rules
```

For a full list of commands, please see pfctl(8).

[Previous: Runtime Options] [Contents] [Next: Packet Queueing and Prioritization]

---

www@openbsd.org

$OpenBSD: anchors.html,v 1.28 2010/05/19 13:25:16 sthen Exp $

OpenBSD

[Previous: Anchors] [Contents] [Next: Address Pools and Load Balancing]

PF: Packet Queueing and Prioritization

Table of Contents

Queueing

To queue something is to store it, in order, while it awaits processing. In a computer network, when data packets are sent out from a host, they enter a queue where they await processing by the operating system. The operating system then decides which queue and which packet(s) from that queue should be processed. The order in which the operating system selects the packets to process can affect network performance. For example, imagine a user running two network applications: SSH and FTP. Ideally, the SSH packets should be processed before the FTP packets because of the time-sensitive nature of SSH; when a key is typed in the SSH client, an immediate response is expected, but an FTP transfer being delayed by a few extra seconds hardly bears any notice. But what happens if the router handling these connections processes a large chunk of packets from the FTP connection before processing the SSH connection? Packets from the SSH connection will remain in the queue (or possibly be dropped by the router if the queue isn't big enough to hold all of the packets) and the SSH session may appear to lag or slow down. By modifying the queueing strategy being used, network bandwidth can be shared fairly between different applications, users, and computers.

Note that queueing is only useful for packets in the *outbound* direction. Once a packet arrives on an interface in the inbound direction it's already too late to queue it -- it's already consumed network bandwidth to get to the interface that just received it. The only solution is to enable queueing on the adjacent router or, if the host that received the packet is acting as a router, to enable queueing on the internal interface where packets exit the router.

Schedulers

The scheduler is what decides which queues to process and in what order. By default, OpenBSD uses a First In First Out (FIFO) scheduler. A FIFO queue works like the line-up at a supermarket's checkout -- the first item into the queue is the first processed. As new packets arrive they are added to the end of the queue. If the queue becomes full, and here the analogy with the supermarket stops, newly arriving packets are dropped. This is known as tail-drop.

OpenBSD supports two additional schedulers:

- Class Based Queueing
- Priority Queueing

Class Based Queueing

Class Based Queueing (CBQ) is a queueing algorithm that divides a network connection's bandwidth among multiple queues or classes. Each queue then has traffic assigned to it based on source or destination address, port number, protocol, etc. A queue may optionally be configured to borrow bandwidth from its parent queue if the parent is being under-utilized. Queues are also given a priority such that those containing interactive traffic, such as SSH, can have their packets processed ahead of queues containing bulk traffic, such as FTP.

CBQ queues are arranged in an hierarchical manner. At the top of the hierarchy is the root queue which defines the total amount of bandwidth available. Child queues are created under the root queue, each of which can be assigned some portion of the root queue's bandwidth. For example, queues might be defined as follows:

```
Root Queue (2Mbps)
    Queue A (1Mbps)
    Queue B (500Kbps)
    Queue C (500Kbps)
```

In this case, the total available bandwidth is set to 2 megabits per second (Mbps). This bandwidth is then split among three child queues.

The hierarchy can further be expanded by defining queues within queues. To split bandwidth equally among different users and also classify their traffic so that certain protocols don't starve others for bandwidth, a queueing structure like this might be defined:

```
Root Queue (2Mbps)
    UserA (1Mbps)
        ssh (50Kbps)
        bulk (950Kbps)
    UserB (1Mbps)
        audio (250Kbps)
        bulk (750Kbps)
            http (100Kbps)
```

>>> other (650Kbps)

Note that at each level the sum of the bandwidth assigned to each of the queues is not more than the bandwidth assigned to the parent queue.

A queue can be configured to borrow bandwidth from its parent if the parent has excess bandwidth available due to it not being used by the other child queues. Consider a queueing setup like this:

> Root Queue (2Mbps)
>> UserA (1Mbps)
>>> ssh (100Kbps)
>>> ftp (900Kbps, borrow)
>>
>> UserB (1Mbps)

If traffic in the `ftp` queue exceeds 900Kbps and traffic in the `UserA` queue is less than 1Mbps (because the `ssh` queue is using less than its assigned 100Kbps), the `ftp` queue will borrow the excess bandwidth from `UserA`. In this way the `ftp` queue is able to use more than its assigned bandwidth when it faces overload. When the `ssh` queue increases its load, the borrowed bandwidth will be returned.

CBQ assigns each queue a priority level. Queues with a higher priority are preferred during congestion over queues with a lower priority as long as both queues share the same parent (in other words, as long as both queues are on the same branch in the hierarchy). Queues with the same priority are processed in a round-robin fashion. For example:

> Root Queue (2Mbps)
>> UserA (1Mbps, priority 1)
>>> ssh (100Kbps, priority 5)
>>> ftp (900Kbps, priority 3)
>>
>> UserB (1Mbps, priority 1)

CBQ will process the `UserA` and `UserB` queues in a round-robin fashion -- neither queue will be preferred over the other. During the time when the `UserA` queue is being processed, CBQ will also process its child queues. In this case, the `ssh` queue has a higher priority and will be given preferential treatment over the `ftp` queue if the network is congested. Note how the `ssh` and `ftp` queues do not have their priorities compared to the `UserA` and `UserB` queues because they are not all on the same branch in the hierarchy.

For a more detailed look at the theory behind CBQ, please see References on CBQ.

Priority Queueing

Priority Queueing (PRIQ) assigns multiple queues to a network interface with each queue being given a priority level. A queue with a higher priority is *always* processed ahead of a queue with a lower priority. If two or more queues are assigned the same priority then those queues are processed in a round-robin fashion.

The queueing structure in PRIQ is flat -- you cannot define queues within queues. The root queue is defined, which sets the total amount of bandwidth that is available, and then sub queues are defined under the root. Consider the following example:

```
Root Queue (2Mbps)
    Queue A (priority 1)
    Queue B (priority 2)
    Queue C (priority 3)
```

The root queue is defined as having 2Mbps of bandwidth available to it and three subqueues are defined. The queue with the highest priority (the highest priority number) is served first. Once all the packets in that queue are processed, or if the queue is found to be empty, PRIQ moves onto the queue with the next highest priority. Within a given queue, packets are processed in a First In First Out (FIFO) manner.

It is important to note that when using PRIQ you must plan your queues very carefully. Because PRIQ *always* processes a higher priority queue before a lower priority one, it's possible for a high priority queue to cause packets in a lower priority queue to be delayed or dropped if the high priority queue is receiving a constant stream of packets.

Random Early Detection

Random Early Detection (RED) is a congestion avoidance algorithm. Its job is to avoid network congestion by making sure that the queue doesn't become full. It does this by continually calculating the average length (size) of the queue and comparing it to two thresholds, a minimum threshold and a maximum threshold. If the average queue size is below the minimum threshold then no packets will be dropped. If the average is above the maximum threshold then *all* newly arriving packets will be dropped. If the average is between the threshold values then packets are dropped based on a probability calculated from the average queue size. In other words, as the average queue size approaches the maximum threshold, more and more packets are dropped. When dropping packets, RED randomly chooses which connections to drop packets from. Connections using larger amounts of bandwidth have a higher probability of having their packets dropped.

RED is useful because it avoids a situation known as global synchronization and it is able to accommodate bursts of traffic. Global synchronization refers to a loss of total throughput due to packets being dropped from several connections at the same time. For example, if congestion occurs at a router carrying traffic for 10 FTP connections and packets from all (or most) of these connections are dropped (as is the case with FIFO queueing), overall throughput will drop sharply. This isn't an ideal situation because it causes all of the FTP connections to reduce their throughput and also means that the network is no longer being used to its maximum potential. RED avoids this by randomly choosing which connections to drop packets from instead of choosing all of them. Connections using large amounts of bandwidth have a higher chance of their packets being dropped. In this way, high bandwidth connections will be throttled back, congestion will be avoided, and sharp losses of overall throughput will not occur. In addition, RED is able to handle bursts of traffic because it starts to drop packets *before* the queue becomes full. When a burst of traffic comes through there will be enough space in the queue to hold the new packets.

RED should only be used when the transport protocol is capable of responding to congestion indicators from the network. In most cases this means RED should be used to queue TCP traffic and not UDP or ICMP traffic.

For a more detailed look at the theory behind RED, please see References on RED.

Explicit Congestion Notification

Explicit Congestion Notification (ECN) works in conjunction with RED to notify two hosts communicating over the network of any congestion along the communication path. It does this by enabling RED to set a flag in the packet header instead of dropping the packet. Assuming the sending host has support for ECN, it can then read this flag and throttle back its network traffic accordingly.

For more information on ECN, please refer to RFC 3168.

Configuring Queueing

Since OpenBSD 3.0 the Alternate Queueing (ALTQ) queueing implementation has been a part of the base system. Starting with OpenBSD 3.3 ALTQ has been integrated into PF. OpenBSD's ALTQ implementation supports the Class Based Queueing (CBQ) and Priority Queueing (PRIQ) schedulers. It also supports Random Early Detection (RED) and Explicit Congestion Notification (ECN).

Because ALTQ has been merged with PF, PF must be enabled for queueing to work. Instructions on how to enable PF can be found in Getting Started.

Queueing is configured in `pf.conf`. There are two types of directives that are used to configure queueing:

- `altq on` - enables queueing on an interface, defines which scheduler to use, and creates the root queue
- `queue` - defines the properties of a child queue

The syntax for the `altq on` directive is:

```
altq on interface scheduler bandwidth bw qlimit qlim \
   tbrsize size queue { queue_list }
```

- *`interface`* - the network interface to activate queueing on.
- *`scheduler`* - the queueing scheduler to use. Possible values are `cbq` and `priq`. Only one scheduler may be active on an interface at a time.
- *`bw`* - the total amount of bandwidth available to the scheduler. This may be specified as an absolute value using the suffixes `b`, `Kb`, `Mb`, and `Gb` to represent bits, kilobits, megabits, and gigabits per second, respectively or as a percentage of the *`interface`* bandwidth.
- *`qlim`* - the maximum number of packets to hold in the queue. This parameter is optional. The default is 50.
- *`size`* - the size of the token bucket regulator in bytes. If not specified, the size is set based on the *`interface`* bandwidth.
- *`queue_list`* - a list of child queues to create under the root queue.

For example:

```
altq on fxp0 cbq bandwidth 2Mb queue { std, ssh, ftp }
```

This enables CBQ on the `fxp0` interface. The total bandwidth available is set to 2Mbps. Three child queues are defined: `std`, `ssh`, and `ftp`.

The syntax for the `queue` directive is:

```
queue name [on interface] bandwidth bw [priority pri] [qlimit
qlim] \
   scheduler ( sched_options ) { queue_list }
```

- *`name`* - the name of the queue. This must match the name of one of the queues defined in the `altq on` directive's *`queue_list`*. For `cbq` it can also match the name of a queue in a previous `queue` directive's *`queue_list`*. Queue names must be no longer than 15 characters.
- *`interface`* - the network interface that the queue is valid on. This value is optional, and when not specified, will make the queue valid on all interfaces.
- *`bw`* - the total amount of bandwidth available to the queue. This may be specified as an absolute value using the suffixes `b`, `Kb`, `Mb`, and `Gb` to represent bits, kilobits, megabits, and gigabits per second, respectively or as a percentage of the parent queue's bandwidth. This parameter is only applicable when using the `cbq` scheduler. If not specified, the default is 100% of the parent queue's bandwidth.
- *`pri`* - the priority of the queue. For `cbq` the priority range is 0 to 7 and for `priq` the range is 0 to 15. Priority 0 is the lowest priority. When not specified, a default of 1 is used.
- *`qlim`* - the maximum number of packets to hold in the queue. When not specified, a default of 50 is used.
- *`scheduler`* - the scheduler being used, either `cbq` or `priq`. Must be the same as the root queue.
- *`sched_options`* - further options may be passed to the scheduler to control its behavior:
  - `default` - defines a default queue where all packets not matching any other queue will be queued. Exactly one default queue is required.
  - `red` - enables Random Early Detection (RED) on this queue.
  - `rio` - enables RED with IN/OUT. In this mode, RED will maintain multiple average queue lengths and multiple threshold values, one for each IP Quality of Service level.
  - `ecn` - enables Explicit Congestion Notification (ECN) on this queue. `Ecn` implies `red`.
  - `borrow` - the queue can borrow bandwidth from its parent. This can only be specified when using the `cbq` scheduler.
- *`queue_list`* - a list of child queues to create under this queue. A *`queue_list`* may only be defined when using the `cbq` scheduler.

Continuing with the example above:

```
queue std bandwidth 50% cbq(default)
queue ssh bandwidth 25% { ssh_login, ssh_bulk }
  queue ssh_login bandwidth 25% priority 4 cbq(ecn)
  queue ssh_bulk bandwidth 75% cbq(ecn)
queue ftp bandwidth 500Kb priority 3 cbq(borrow red)
```

Here the parameters of the previously defined child queues are set. The `std` queue is assigned a bandwidth of 50% of the root queue's bandwidth (or 1Mbps) and is set as the default queue. The `ssh` queue is assigned 25% of the root queue's bandwidth (500kb) and also contains two child queues, `ssh_login` and `ssh_bulk`. The `ssh_login` queue is given a higher priority than `ssh_bulk` and both have ECN enabled. The `ftp` queue is assigned a bandwidth of 500Kbps and given a priority of 3. It can also borrow bandwidth when extra is available and has RED enabled.

NOTE: Each child queue definition has its bandwidth specified. Without specifying the bandwidth, PF will give the queue 100% of the parent queue's bandwidth. In this situation, that would cause an error when the rules are loaded since if there's a queue with 100% of the bandwidth, no other queue can be defined at that level since there is no free bandwidth to allocate to it.

Assigning Traffic to a Queue

To assign traffic to a queue, the `queue` keyword is used in conjunction with PF's filter rules. For example, consider a set of filtering rules containing a line such as:

```
pass out on fxp0 proto tcp to port 22
```

Packets matching that rule can be assigned to a specific queue by using the `queue` keyword:

```
pass out on fxp0 proto tcp to port 22 queue ssh
```

When a state table entry is created by this rule, PF will record the queue in the state table entry; this will be used for other packets permitted by the entry:

```
pass in on fxp0 proto tcp to port 80 queue http
```

With this rule, packets traveling back out `fxp0` that match the stateful connection will end up in the `http` queue. Note that even though the `queue` keyword is being used on a rule filtering incoming traffic, the goal is to specify a queue for the corresponding outgoing traffic; the above rule does not queue incoming packets.

When using the `queue` keyword with `block` directives, any resulting TCP RST or ICMP Unreachable packets are assigned to the specified queue.

Note that queue designation can happen on an interface other than the one defined in the `altq on` directive:

```
altq on fxp0 cbq bandwidth 2Mb queue { std, ftp }
queue std bandwidth 500Kb cbq(default)
queue ftp bandwidth 1.5Mb

pass in on dc0 proto tcp to port 21 queue ftp
```

Queueing is enabled on `fxp0` but the designation takes place on `dc0`. If packets matching the `pass` rule (or the state created by this rule) exit from interface `fxp0`, they will be queued in the `ftp` queue. This type of queueing can be very useful on routers.

Normally only one queue name is given with the `queue` keyword, but if a second name is specified that queue will be used for packets with a Type of Service (ToS) of low-delay and for TCP ACK packets with no data payload. A good example of this is found when using SSH. SSH login sessions will set the ToS to low-delay while SCP and SFTP sessions will not. PF can use this information to queue packets belonging to a login connection in a different queue than non-login connections. This can be useful to prioritize login connection packets over file transfer packets.

```
pass out on fxp0 from any to any port 22 queue(ssh_bulk, ssh_login)
```

This assigns packets belonging to SSH login connections to the `ssh_login` queue and packets belonging to SCP and SFTP connections to the `ssh_bulk` queue. SSH login connections will then have their packets processed ahead of SCP and SFTP connections because the `ssh_login` queue has a higher priority.

Assigning TCP ACK packets to a higher priority queue is useful on asymmetric connections, that is, connections that have different upload and download bandwidths such as ADSL lines. With an ADSL line, if the upload channel is being maxed out and a download is started, the download will suffer because the TCP ACK packets it needs to send will run into congestion when they try to pass through the upload channel. Testing has shown that to achieve the best results, the bandwidth on the upload queue should be set to a value less than what the connection is capable of. For instance, if an ADSL line has a max upload of 640Kbps, setting the root queue's `bandwidth` to a value such as 600Kb should result in better performance. Trial and error will yield the best `bandwidth` setting.

Example #1: Small, Home Network

```
    [ Alice ]    [ Charlie ]
        |             |                              ADSL
     ---+-----+-------+------ dc0 [ OpenBSD ] fxp0 -------- ( Internet )
              |
           [ Bob ]
```

In this example, OpenBSD is being used on an Internet gateway for a small home network with three workstations. The gateway is performing packet filtering and NAT duties. The Internet connection is via an ADSL line running at 2Mbps down and 640Kbps up.

The queueing policy for this network:

- Reserve 80Kbps of download bandwidth for Bob so he can play his online games without being lagged by Alice or Charlie's downloads. Allow Bob to use more than 80Kbps when it's available.
- Interactive SSH and instant message traffic will have a higher priority than regular traffic.
- DNS queries and replies will have the second highest priority.
- Outgoing TCP ACK packets will have a higher priority than all other outgoing traffic.

Below is the ruleset that meets this network policy. Note that only the `pf.conf` directives that apply directly to the above policy are present.

```
# enable queueing on the external interface to control traffic going to
# the Internet. use the priq scheduler to control only priorities. set
# the bandwidth to 610Kbps to get the best performance out of the TCP
# ACK queue.

altq on fxp0 priq bandwidth 610Kb queue { std_out, ssh_im_out, dns_out, \
        tcp_ack_out }

# define the parameters for the child queues.
# std_out      - the standard queue. any filter rule below that does not
#                explicitly specify a queue will have its traffic added
#                to this queue.
# ssh_im_out   - interactive SSH and various instant message traffic.
# dns_out      - DNS queries.
# tcp_ack_out  - TCP ACK packets with no data payload.

queue std_out     priq(default)
queue ssh_im_out  priority 4 priq(red)
queue dns_out     priority 5
queue tcp_ack_out priority 6

# enable queueing on the internal interface to control traffic coming in
# from the Internet. use the cbq scheduler to control bandwidth. max
# bandwidth is 2Mbps.

altq on dc0 cbq bandwidth 2Mb queue { std_in, ssh_im_in, dns_in, bob_in }

# define the parameters for the child queues.
# std_in      - the standard queue. any filter rule below that does not
#               explicitly specify a queue will have its traffic added
#               to this queue.
# ssh_im_in   - interactive SSH and various instant message traffic.
# dns_in      - DNS replies.
# bob_in      - bandwidth reserved for Bob's workstation. allow him to
#               borrow.

queue std_in    bandwidth 1.6Mb cbq(default)
queue ssh_im_in bandwidth 200Kb priority 4
queue dns_in    bandwidth 120Kb priority 5
queue bob_in    bandwidth 80Kb cbq(borrow)


# ... in the filtering section of pf.conf ...

alice         = "192.168.0.2"
bob           = "192.168.0.3"
charlie       = "192.168.0.4"
```

```
local_net     = "192.168.0.0/24"
ssh_ports     = "{ 22 2022 }"
im_ports      = "{ 1863 5190 5222 }"

# filter rules for fxp0 inbound
block in on fxp0 all

# filter rules for fxp0 outbound
block out on fxp0 all
pass  out on fxp0 inet proto tcp from (fxp0) queue(std_out, tcp_ack_out)
pass  out on fxp0 inet proto { udp icmp } from (fxp0)
pass  out on fxp0 inet proto { tcp udp } from (fxp0) to port domain \
        queue dns_out
pass  out on fxp0 inet proto tcp from (fxp0) to port $ssh_ports \
        queue(std_out, ssh_im_out)
pass  out on fxp0 inet proto tcp from (fxp0) to port $im_ports \
        queue(ssh_im_out, tcp_ack_out)

# filter rules for dc0 inbound
block in on dc0 all
pass  in on dc0 from $local_net

# filter rules for dc0 outbound
block out on dc0 all
pass  out on dc0 to $local_net
pass  out on dc0 proto { tcp udp } from port domain to $local_net \
        queue dns_in
pass  out on dc0 proto tcp from port $ssh_ports to $local_net \
        queue(std_in, ssh_im_in)
pass  out on dc0 proto tcp from port $im_ports to $local_net \
        queue ssh_im_in
pass  out on dc0 to $bob queue bob_in
```

Example #2: Company Network

```
  ( IT Dept )  [ Boss's PC ]
       |          |                                   T1
     --+----+-----+---------- dc0 [ OpenBSD ] fxp0 -------- ( Internet )
            |                           fxp1
         [ COMP1 ]    [ WWW ]         /
                         |           /
                       --+----------'
```

In this example, the OpenBSD host is acting as a firewall for a company network. The company runs a WWW server in the DMZ portion of their network where customers upload their websites via FTP. The IT department has their own subnet connected to the main network, and the boss has a PC on his desk that's used for email and surfing the web. The connection to the Internet is via a T1 line running at 1.5Mbps in both directions. All other network segments are using Fast Ethernet (100Mbps).

The network administrator has decided on the following policy:

- Limit all traffic between the WWW server and the Internet to 500Kbps in each direction.
    - Allot 250Kbps to HTTP traffic.
    - Allot 250Kbps to "other" traffic (i.e., non-HTTP traffic)
    - Allow either queue to borrow up to the full 500Kbps.
    - Give HTTP traffic between the WWW server and the Internet a higher priority than other traffic between the WWW server and the Internet (such as FTP uploads).
- Traffic between the WWW server and the internal network can use up to the full 100Mbps that the network offers.
- Reserve 500Kbps for the IT Dept network so they can download the latest software updates in a timely manner. They should be able to use more than 500Kbps when extra bandwidth is available.
- Give traffic between the boss's PC and the Internet a higher priority than other traffic to/from the Internet.

Below is the ruleset that meets this network policy. Note that only the `pf.conf` directives that apply directly to the above policy are present; `nat`, `rdr`, options, etc., are not shown.

```
# enable queueing on the external interface to queue packets going out
# to the Internet. use the cbq scheduler so that the bandwidth use of
# each queue can be controlled. the max outgoing bandwidth is 1.5Mbps.

altq on fxp0 cbq bandwidth 1.5Mb queue { std_ext, www_ext, boss_ext }

# define the parameters for the child queues.
# std_ext        - the standard queue. also the default queue for
#                  outgoing traffic on fxp0.
# www_ext        - container queue for WWW server queues. limit to
#                  500Kbps.
#   www_ext_http - http traffic from the WWW server; higher priority.
#   www_ext_misc - all non-http traffic from the WWW server.
# boss_ext       - traffic coming from the boss's computer.

queue std_ext        bandwidth 500Kb cbq(default borrow)
queue www_ext        bandwidth 500Kb { www_ext_http, www_ext_misc }
  queue www_ext_http bandwidth 50% priority 3 cbq(red borrow)
  queue www_ext_misc bandwidth 50% priority 1 cbq(borrow)
queue boss_ext       bandwidth 500Kb priority 3 cbq(borrow)

# enable queueing on the internal interface to control traffic coming
# from the Internet or the DMZ. use the cbq scheduler to control the
# bandwidth of each queue. bandwidth on this interface is set to the
# maximum. traffic coming from the DMZ will be able to use all of this
# bandwidth while traffic coming from the Internet will be limited to
# 1.0Mbps (because 0.5Mbps (500Kbps) is being allocated to fxp1).

altq on dc0 cbq bandwidth 100% queue { net_int, www_int }

# define the parameters for the child queues.
# net_int    - container queue for traffic from the Internet. bandwidth
#              is 1.0Mbps.
#   std_int  - the standard queue. also the default queue for outgoing
#              traffic on dc0.
#   it_int   - traffic to the IT Dept network; reserve them 500Kbps.
#   boss_int - traffic to the boss's PC; assign a higher priority.
# www_int    - traffic from the WWW server in the DMZ; full speed.

queue net_int    bandwidth 1.0Mb { std_int, it_int, boss_int }
  queue std_int  bandwidth 250Kb cbq(default borrow)
  queue it_int   bandwidth 500Kb cbq(borrow)
  queue boss_int bandwidth 250Kb priority 3 cbq(borrow)
queue www_int    bandwidth 99Mb cbq(red borrow)

# enable queueing on the DMZ interface to control traffic destined for
# the WWW server. cbq will be used on this interface since detailed
# control of bandwidth is necessary. bandwidth on this interface is set
# to the maximum. traffic from the internal network will be able to use
# all of this bandwidth while traffic from the Internet will be limited
# to 500Kbps.

altq on fxp1 cbq bandwidth 100% queue { internal_dmz, net_dmz }

# define the parameters for the child queues.
# internal_dmz   - traffic from the internal network.
# net_dmz        - container queue for traffic from the Internet.
#   net_dmz_http - http traffic; higher priority.
#   net_dmz_misc - all non-http traffic. this is also the default queue.

queue internal_dmz   bandwidth 99Mb cbq(borrow)
queue net_dmz        bandwidth 500Kb { net_dmz_http, net_dmz_misc }
  queue net_dmz_http bandwidth 50% priority 3 cbq(red borrow)
  queue net_dmz_misc bandwidth 50% priority 1 cbq(default borrow)



# ... in the filtering section of pf.conf ...

main_net  = "192.168.0.0/24"
it_net    = "192.168.1.0/24"
int_nets  = "{ 192.168.0.0/24, 192.168.1.0/24 }"
```

```
dmz_net   = "10.0.0.0/24"

boss      = "192.168.0.200"
wwwserv   = "10.0.0.100"

# default deny
block on { fxp0, fxp1, dc0 } all

# filter rules for fxp0 inbound
pass in on fxp0 proto tcp from any to $wwwserv port { 21, \
        > 49151 } queue www_ext_misc
pass in on fxp0 proto tcp from any to $wwwserv port 80 queue www_ext_http

# filter rules for fxp0 outbound
pass out on fxp0 from $int_nets
pass out on fxp0 from $boss queue boss_ext

# filter rules for dc0 inbound
pass in on dc0 from $int_nets
pass in on dc0 from $it_net queue it_int
pass in on dc0 from $boss queue boss_int
pass in on dc0 proto tcp from $int_nets to $wwwserv port { 21, 80, \
        > 49151 } queue www_int

# filter rules for dc0 outbound
pass out on dc0 from dc0 to $int_nets

# filter rules for fxp1 inbound
pass in on fxp1 proto { tcp, udp } from $wwwserv to port 53

# filter rules for fxp1 outbound
pass out on fxp1 proto tcp to $wwwserv port { 21, \
        > 49151 } queue net_dmz_misc
pass out on fxp1 proto tcp to $wwwserv port 80 queue net_dmz_http
pass out on fxp1 proto tcp from $int_nets to $wwwserv port { 80, \
        21, > 49151 } queue internal_dmz
```

[Previous: Anchors] [Contents] [Next: Address Pools and Load Balancing]

www@openbsd.org

$OpenBSD: queueing.html,v 1.39 2010/05/19 13:25:16 sthen Exp $

OpenBSD

[Previous: Packet Queueing and Prioritization] [Contents] [Next: Packet Tagging]

PF: Address Pools and Load Balancing

Table of Contents

Introduction

An address pool is a supply of two or more addresses whose use is shared among a group of users. An address pool can be specified as the target address in `nat-to`, `rdr-to`,`route-to`, `reply-to`, and `dup-to` filter options.

There are four methods for using an address pool:

- `bitmask` - grafts the network portion of the pool address over top of the address that is being modified (source address for `nat-to` rules, destination address for `rdr-to` rules). Example: if the address pool is 192.0.2.1/24 and the address being modified is 10.0.0.50, then the resulting address will be 192.0.2.50. If the address pool is 192.0.2.1/25 and the address being modified is 10.0.0.130, then the resulting address will be 192.0.2.2.
- `random` - randomly selects an address from the pool.
- `source-hash` - uses a hash of the source address to determine which address to use from the pool. This method ensures that a given source address is always mapped to the same pool address. The key that is fed to the hashing algorithm can optionally be specified after the `source-hash` keyword in hex format or as a string. By default, pfctl(8) will generate a random key every time the ruleset is loaded.
- `round-robin` - loops through the address pool in sequence. This is the default method and also the only method allowed when the address pool is specified using a table.

Except for the `round-robin` method, the address pool must be expressed as a CIDR (Classless Inter-Domain Routing) network block. The `round-robin` method will accept multiple individual addresses using a list or table.

The `sticky-address` option can be used with the `random` and `round-robin` pool types to ensure that a particular source address is always mapped to the same redirection address.

NAT Address Pool

An address pool can be used as the translation address in `nat-to` rules. Connections will have their source address translated to an address from the pool based on the method chosen. This can be useful in situations where PF is performing NAT for a very large network. Since the number of NATed connections per translation address is limited, adding additional translation addresses will allow the NAT gateway to scale to serve a larger number of users.

In this example a pool of two addresses is being used to translate outgoing packets. For each outgoing connection PF will rotate through the addresses in a round-robin manner.

```
match out on $ext_if inet nat-to { 192.0.2.5, 192.0.2.10 }
```

One drawback with this method is that successive connections from the same internal address will not always be translated to the same translation address. This can cause interference, for example, when browsing websites that track user logins based on IP address. An alternate approach is to use the `source-hash` method so that each internal address is always translated to the same translation address. To do this, the address pool must be a CIDR network block.

```
match out on $ext_if inet nat-to 192.0.2.4/31 source-hash
```

This rule uses the address pool 192.0.2.4/31 (192.0.2.4 - 192.0.2.5) as the translation address for outgoing packets. Each internal address will always be translated to the same translation address because of the `source-hash` keyword.

Load Balance Incoming Connections

Address pools can also be used to load balance incoming connections. For example, incoming web server connections can be distributed across a web server farm:

```
web_servers = "{ 10.0.0.10, 10.0.0.11, 10.0.0.13 }"

match in on $ext_if proto tcp to port 80 rdr-to $web_servers \
    round-robin sticky-address
```

Successive connections will be redirected to the web servers in a round-robin manner with connections from the same source being sent to the same web server. This "sticky connection" will exist as long as there are states that refer to this connection. Once the states expire, so will the sticky connection. Further connections from that host will be redirected to the next web server in the round robin.

Load Balance Outgoing Traffic

Address pools can be used in combination with the `route-to` filter option to load balance two or more Internet connections when a proper multi-path routing protocol (like BGP4) is unavailable. By using `route-to` with a `round-robin` address pool, outbound connections can be evenly distributed among multiple outbound paths.

One additional piece of information that's needed to do this is the IP address of the adjacent router on each Internet connection. This is fed to the `route-to` option to control the destination of outgoing packets.

The following example balances outgoing traffic across two Internet connections:

```
lan_net = "192.168.0.0/24"
int_if  = "dc0"
ext_if1 = "fxp0"
ext_if2 = "fxp1"
ext_gw1 = "68.146.224.1"
ext_gw2 = "142.59.76.1"

pass in on $int_if from $lan_net \    route-to { ($ext_if1
$ext_gw1), ($ext_if2 $ext_gw2) } \    round-robin
```

The `route-to` option is used on traffic coming *in* on the *internal* interface to specify the outgoing network interfaces that traffic will be balanced across along with their respective gateways. Note that the `route-to` option must be present on *each* filter rule that traffic is to be balanced for (it cannot be used with `match` rules).

To ensure that packets with a source address belonging to `$ext_if1` are always routed to `$ext_gw1` (and similarly for `$ext_if2` and `$ext_gw2`), the following two lines should be included in the ruleset:

```
pass out on $ext_if1 from $ext_if2 \
   route-to ($ext_if2 $ext_gw2)
pass out on $ext_if2 from $ext_if1 \
   route-to ($ext_if1 $ext_gw1)
```

Finally, NAT can also be used on each outgoing interface:

```
match out on $ext_if1 from $lan_net nat-to ($ext_if1)
match out on $ext_if2 from $lan_net nat-to ($ext_if2)
```

A complete example that load balances outgoing traffic might look something like this:

```
lan_net = "192.168.0.0/24"
int_if  = "dc0"
ext_if1 = "fxp0"
ext_if2 = "fxp1"
ext_gw1 = "68.146.224.1"
ext_gw2 = "142.59.76.1"

#  nat outgoing connections on each internet interface
match out on $ext_if1 from $lan_net nat-to ($ext_if1)
match out on $ext_if2 from $lan_net nat-to ($ext_if2)

#  default deny
block in
block out

#  pass all outgoing packets on internal interface
pass out on $int_if to $lan_net
#  pass in quick any packets destined for the gateway itself
pass in quick on $int_if from $lan_net to $int_if
#  load balance outgoing traffic from internal network.
pass in on $int_if from $lan_net \
    route-to { ($ext_if1 $ext_gw1), ($ext_if2 $ext_gw2) } \
    round-robin
#  keep https traffic on a single connection; some web applications,
#  especially "secure" ones, don't allow it to change mid-session
pass in on $int_if proto tcp from $lan_net to port https \
    route-to ($ext_if1 $ext_gw1)

#  general "pass out" rules for external interfaces
pass out on $ext_if1
pass out on $ext_if2

#  route packets from any IPs on $ext_if1 to $ext_gw1 and the same for
#  $ext_if2 and $ext_gw2
pass out on $ext_if1 from $ext_if2 route-to ($ext_if2 $ext_gw2)
pass out on $ext_if2 from $ext_if1 route-to ($ext_if1 $ext_gw1)
```

[Previous: Packet Queueing and Prioritization] [Contents] [Next: Packet Tagging]

www@openbsd.org

$OpenBSD: pools.html,v 1.26 2010/05/19 13:25:16 sthen Exp $

[Previous: Address Pools and Load Balancing] [Contents] [Next: Logging]

PF: Packet Tagging (Policy Filtering)

Table of Contents

Introduction

Packet tagging is a way of marking packets with an internal identifier that can later be used in filter and translation rule criteria. With tagging, it's possible to do such things as create "trusts" between interfaces and determine if packets have been processed by translation rules. It's also possible to move away from rule-based filtering and to start doing policy-based filtering.

Assigning Tags to Packets

To add a tag to a packet, use the `tag` keyword:

```
pass in on $int_if all tag INTERNAL_NET keep state
```

The tag `INTERNAL_NET` will be added to any packet which matches the above rule.

A tag can also be assigned using a macro. For instance:

```
name = "INTERNAL_NET"
pass in on $int_if all tag $name
```

There are a set of predefined macros which can also be used.

- `$if` - The interface
- `$srcaddr` - Source IP address

- `$dstaddr` - Destination IP address
- `$srcport` - The source port specification
- `$dstport` - The destination port specification
- `$proto` - The protocol
- `$nr` - The rule number

These macros are expanded at ruleset load time and NOT at runtime.

Tagging follows these rules:

- Tags are "sticky". Once a tag is applied to a packet by a matching rule it is never removed. It can, however, be replaced with a different tag.
- Because of a tag's "stickiness", a packet can have a tag even if the last matching rule doesn't use the `tag` keyword.
- A packet is only ever assigned a maximum of one tag at a time.
- Tags are *internal* identifiers. Tags are not sent out over the wire.
- Tag names can be up to 63 characters long. In OpenBSD 4.0 and earlier, tag names are limited to 15 characters.

Take the following ruleset as an example.

```
(1) pass in on $int_if tag INT_NET
(2) pass in quick on $int_if proto tcp to port 80 tag INT_NET_HTTP
(3) pass in quick on $int_if from 192.168.1.5
```

- Packets coming in on `$int_if` will be assigned a tag of `INT_NET` by rule #1.
- TCP packets coming in on `$int_if` and destined for port 80 will first be assigned a tag of `INT_NET` by rule #1. That tag will then be replaced with the `INT_NET_HTTP` tag by rule #2.
- Packets coming in on `$int_if` from 192.168.1.5 will be tagged one of two ways. If the packet is destined for TCP port 80 it will match rule #2 and be tagged with `INT_NET_HTTP`. Otherwise, the packet will match rule #3 but will be tagged with `INT_NET`. Because the packet matches rule #1, the `INT_NET` tag is applied and is not removed unless a subsequently matching rule specifies a tag (this is the "stickiness" of a tag).

Checking for Applied Tags

To check for previously applied tags, use the `tagged` keyword:

```
pass out on $ext_if tagged INT_NET
```

Outgoing packets on `$ext_if` must be tagged with the `INT_NET` tag in order to match the above rule. Inverse matching can also be done by using the `!` operator:

```
pass out on $ext_if ! tagged WIFI_NET
```

Policy Filtering

Policy filtering takes a different approach to writing a filter ruleset. A policy is defined which sets the rules for what types of traffic is passed and what types are blocked. Packets are then classified into the policy based on the traditional criteria of source/destination IP address/port, protocol, etc. For example, examine the following firewall policy:

- Traffic from the internal LAN to the Internet is permitted (LAN_INET) and must be translated (LAN_INET_NAT)
- Traffic from the internal LAN to the DMZ is permitted (LAN_DMZ)
- Traffic from the Internet to servers in the DMZ is permitted (INET_DMZ)
- Traffic from the Internet that's being redirected to spamd(8) is permitted (SPAMD)
- All other traffic is blocked

Note how the policy covers *all* traffic that will be passing through the firewall. The item in parenthesis indicates the tag that will be used for that policy item.

Rules now need to be written to classify packets into the policy.

```
block all
pass out on $ext_if tag LAN_INET_NAT tagged LAN_INET nat-to
($ext_if)
pass in on $int_if from $int_net tag LAN_INET
pass in on $int_if from $int_net to $dmz_net tag LAN_DMZ
pass in on $ext_if proto tcp to $www_server port 80 tag INET_DMZ
pass in on $ext_if proto tcp from <spamd> to port smtp \
   tag SPAMD rdr-to 127.0.0.1 port 8025
```

Now the rules that define the policy are set.

```
pass in  quick on $ext_if tagged SPAMD
pass out quick on $ext_if tagged LAN_INET_NAT
pass out quick on $dmz_if tagged LAN_DMZ
pass out quick on $dmz_if tagged INET_DMZ
```

Now that the whole ruleset is setup, changes are a matter of modifying the classification rules. For example, if a POP3/SMTP server is added to the DMZ, it will be necessary to add classification rules for POP3 and SMTP traffic, like so:

```
mail_server = "192.168.0.10"
...
pass in on $ext_if proto tcp to $mail_server port { smtp, pop3 } \
   tag INET_DMZ
```

Email traffic will now be passed as part of the INET_DMZ policy entry.

The complete ruleset:

```
# macros
int_if  = "dc0"
dmz_if  = "dc1"
ext_if  = "ep0"
int_net = "10.0.0.0/24"
dmz_net = "192.168.0.0/24"
www_server = "192.168.0.5"
mail_server = "192.168.0.10"

table <spamd> persist file "/etc/spammers"

# classification -- classify packets based on the defined firewall
# policy.
block all
pass out on $ext_if tag LAN_INET_NAT tagged LAN_INET nat-to ($ext_if)

pass in on $int_if from $int_net tag LAN_INET

pass in on $int_if from $int_net to $dmz_net tag LAN_DMZ

pass in on $ext_if proto tcp to $www_server port 80 tag INET_DMZ
pass in on $ext_if proto tcp from <spamd> to port smtp \

   tag SPAMD rdr-to 127.0.0.1 port 8025


# policy enforcement -- pass/block based on the defined firewall policy.
pass in  quick on $ext_if tagged SPAMD
pass out quick on $ext_if tagged LAN_INET_NAT
pass out quick on $dmz_if tagged LAN_DMZ
pass out quick on $dmz_if tagged INET_DMZ
```

Tagging Ethernet Frames

Tagging can be performed at the Ethernet level if the machine doing the tagging/filtering is also acting as a bridge(4). By creating bridge(4) filter rules that use the `tag` keyword, PF can be made to filter based on the source or destination MAC address. Bridge(4) rules are created using the brconfig(8) command. Example:

```
# ifconfig bridge0 rule pass in on fxp0 src 0:de:ad:be:ef:0 \
   tag USER1
```

And then in `pf.conf`:

```
    pass in on fxp0 tagged USER1
```

[Previous: Address Pools and Load Balancing] [Contents] [Next: Logging]

www@openbsd.org

$OpenBSD: tagging.html,v 1.19 2010/05/19 13:25:16 sthen Exp $

[Previous: Packet Tagging] [Contents] [Next: Performance]

PF: Logging

Table of Contents

Introduction

When a packet is logged by PF, a copy of the packet header is sent to a pflog(4) interface along with some additional data such as the interface the packet was transiting, the action that PF took (pass or block), etc. The pflog(4) interface allows user-space applications to receive PF's logging data from the kernel. If PF is enabled when the system is booted, the pflogd(8) daemon is started. By default pflogd(8) listens on the `pflog0` interface and writes all logged data to the `/var/log/pflog` file.

Logging Packets

In order to log packets passing through PF, the `log` keyword must be used. The `log` keyword causes all packets that match the rule to be logged. In the case where the rule is creating state, only the first packet seen (the one that causes the state to be created) will be logged.

The options that can be given to the `log` keyword are:

`all`
: Causes all matching packets, not just the initial packet, to be logged. Useful for rules that create state.

`to` *`pflogN`*

Causes all matching packets to be logged to the specified pflog(4) interface. For example, when using spamlogd(8) all SMTP traffic can be logged to a dedicated pflog(4) interface by PF. The spamlogd(8) daemon can then be told to listen on that interface. This keeps the main PF logfile clean of SMTP traffic which otherwise would not need to be logged. Use ifconfig(8) to create pflog(4) interfaces. The default log interface `pflog0` is created automatically.

`user`

Causes the UNIX user-id and group-id that owns the socket that the packet is sourced from/destined to (whichever socket is local) to be logged along with the standard log information.

Options are given in parenthesis after the `log` keyword; multiple options can be separated by a comma or space.

```
    pass in log (all, to pflog1) on $ext_if inet proto tcp to
    $ext_if port 22 keep state
```

Reading a Log File

The log file written by pflogd is in binary format and cannot be read using a text editor. Tcpdump must be used to view the log.

To view the log file:

```
    # tcpdump -n -e -ttt -r /var/log/pflog
```

Note that using tcpdump(8) to watch the pflog file does *not* give a real-time display. A real-time display of logged packets is achieved by using the `pflog0` interface:

```
    # tcpdump -n -e -ttt -i pflog0
```

NOTE: When examining the logs, special care should be taken with tcpdump's verbose protocol decoding (activated via the `-v` command line option). Tcpdump's protocol decoders do not have a perfect security history. At least in theory, a delayed attack could be possible via the partial packet payloads recorded by the logging device. It is recommended practice to move the log files off of the firewall machine before examining them in this way.

Additional care should also be taken to secure access to the logs. By default, pflogd will record 96 bytes of the packet in the log file. Access to the logs could provide partial access to sensitive packet payloads (like telnet(1) or ftp(1) usernames and passwords).

Filtering Log Output

Because pflogd logs in tcpdump binary format, the full range of tcpdump features can be used when reviewing the logs. For example, to only see packets that match a certain port:

```
# tcpdump -n -e -ttt -r /var/log/pflog port 80
```

This can be further refined by limiting the display of packets to a certain host and port combination:

```
# tcpdump -n -e -ttt -r /var/log/pflog port 80 and host
192.168.1.3
```

The same idea can be applied when reading from the `pflog0` interface:

```
# tcpdump -n -e -ttt -i pflog0 host 192.168.4.2
```

Note that this has no impact on which packets are logged to the pflogd log file; the above commands only display packets as they are being logged.

In addition to using the standard tcpdump(8) filter rules, the tcpdump filter language has been extended for reading pflogd output:

- `ip` - address family is IPv4.
- `ip6` - address family is IPv6.
- `on` *`int`* - packet passed through the interface *int*.
- `ifname` *`int`* - same as `on` *`int`*.
- `ruleset` *`name`* - the ruleset/anchor that the packet was matched in.
- `rulenum` *`num`* - the filter rule that the packet matched was rule number *num*.
- `action` *`act`* - the action taken on the packet. Possible actions are `pass` and `block`.
- `reason` *`res`* - the reason that `action` was taken. Possible reasons are `match`, `bad-offset`, `fragment`, `short`, `normalize`, `memory`, `bad-timestamp`, `congestion`, `ip-option`, `proto-cksum`, `state-mismatch`, `state-insert`, `state-limit`, `src-limit`, and `synproxy`.
- `inbound` - packet was inbound.
- `outbound` - packet was outbound.

Example:

```
# tcpdump -n -e -ttt -i pflog0 inbound and action block and
on wi0
```

This display the log, in real-time, of inbound packets that were blocked on the wi0 interface.

Packet Logging Through Syslog

In many situations it is desirable to have the firewall logs available in ASCII format and/or to send them to a remote logging server. All this can be accomplished with a small shell script, some minor changes of the OpenBSD configuration files, and syslogd(8), the logging daemon. Syslogd logs in ASCII and is also able to log to a remote logging server.

Create the following script:

`/etc/pflogrotate`

```
        #!/bin/sh
        PFLOG=/var/log/pflog
        FILE=/var/log/pflog5min.$(date "+%Y%m%d%H%M")
        pkill -ALRM -u root -U root -t - -x pflogd
        if [ -r $PFLOG ] && [ $(stat -f %z $PFLOG) -gt 24 ]; then
           mv $PFLOG $FILE
           pkill -HUP -u root -U root -t - -x pflogd
           tcpdump -n -e -ttt -r $FILE | logger -t pf -p local0.info
           rm $FILE
        fi
```

Edit root's cron job:

```
     # crontab -u root -e
```

Add the following two lines:

```
     # rotate pf log file every 5 minutes
     0-59/5 * * * * /bin/sh /etc/pflogrotate
```

Add the following line to `/etc/syslog.conf`:

```
     local0.info     /var/log/pflog.txt
```

If you also want to log to a remote log server, add the line:

```
     local0.info     @syslogger
```

Make sure host *syslogger* has been defined in the hosts(5) file.

Create the file `/var/log/pflog.txt` to allow syslog to log to that file, and give it the same permissions as the pflog file.

```
     # touch /var/log/pflog.txt
     # chmod 600 /var/log/pflog.txt
```

Make syslogd notice the changes by restarting it:

```
     # kill -HUP $(cat /var/run/syslog.pid)
```

All logged packets are now sent to `/var/log/pflog.txt`. If the second line is added they are sent to the remote logging host *syslogger* as well.

The script `/etc/pflogrotate` now processes and then deletes `/var/log/pflog` so rotation of `pflog` by newsyslog(8) is no longer necessary and should be disabled. However, `/var/log/pflog.txt` replaces `/var/log/pflog` and rotation of it should be activated. Change `/etc/newsyslog.conf` as follows:

```
    #/var/log/pflog       600    3    250     *     ZB "pkill -HUP -u
root -U root -t - -x pflogd"
    /var/log/pflog.txt    600    7    *       24
```

PF will now log in ASCII to `/var/log/pflog.txt`. If so configured in `/etc/syslog.conf`, it will also log to a remote server. The logging is not immediate but it can take up to about 5-6 minutes (the cron job interval) before the logged packets appear in the file.

[Previous: Packet Tagging] [Contents] [Next: Performance]

---

www@openbsd.org

$OpenBSD: logging.html,v 1.41 2010/07/15 00:22:22 sthen Exp $

[Previous: Logging] [Contents] [Next: Issues with FTP]

PF: Performance

"How much bandwidth can PF handle?"
"How much computer do I need to handle my Internet connection?"

There are no easy answers to those questions. For some applications, a 486/66 with a pair of good ISA NICs could filter and NAT close to 5Mbps, but for other applications a much faster machine with much more efficient PCI NICs might end up being insufficient. The real question is not the number of bits per second but rather the number of packets per second and the complexity of the ruleset.

PF performance is determined by several variables:

- Number of packets per second. Almost the same amount of processing needs to be done on a packet with 1500 byte payload as for a packet with a one byte payload. The number of packets per second determines the number of times the state table and, in case of no match there, filter rules have to be evaluated every second, determining the effective demand on the system.
- Performance of your system bus. The ISA bus has a maximum bandwidth of 8MB/sec, and when the processor is accessing it, it has to slow itself to the effective speed of a 80286 running at 8MHz, no matter how fast the processor really is. The PCI bus has a much greater effective bandwidth, and has less impact on the processor.
- Efficiency of your network card. Some network adapters are just more efficient than others. Older rl(4) Realtek 8139 based cards tend to be relatively poor performers (newer re(4)-based Realtek cards are much better), while Intel 21143 (dc(4)) based cards tend to perform very well. For maximum performance, consider using gigabit Ethernet cards, even if not connecting to gigabit networks, as they have much more advanced buffering.
- Complexity and design of your ruleset. The more complex your ruleset, the slower it is. The more packets that are filtered by `keep state` and `quick` rules, the better the performance. The more lines that have to be evaluated for each packet, the lower the performance.
- Barely worth mentioning: CPU and RAM. As PF is a kernel-based process, it will not use swap space. So, if you have enough RAM, it runs, if not, it panics due to pool(9) exhaustion. Huge amounts of RAM are not needed -- 32MB should be plenty for close to 30,000 states, which is a lot of states for a small office or home application. Most users will find a "recycled" computer more than enough for a PF system -- a 300MHz system will move a large number of packets rapidly, at least if backed up with good NICs and a good ruleset.

Will multiple processors help?

PF will only use one processor, so multiple processors (or multiple cores) WILL NOT improve PF performance. HOWEVER, under some circumstances, running the SMP version of OpenBSD (`bsd.mp`) instead of `bsd` will give better performance due to differences in how interrupt handling is done. In many cases, `bsd.mp` will give less performance. IF you are seeing performance problems, experiment with this, most users will never hit any limits to worry about it.

Are there any benchmarks?

People often ask for PF benchmarks. The only benchmark that counts is *your* system performance in *your* environment. A benchmark that doesn't replicate your environment will not properly help you plan your firewall system. The best course of action is to benchmark PF for yourself under the same, or as close as possible to, network conditions that the actual firewall would experience running on the same hardware the firewall would use.

PF is used in some very large, high-traffic applications, and the developers are "power users" of PF. Odds are, it will do very well for you.

[Previous: Logging] [Contents] [Next: Issues with FTP]

www@openbsd.org

$OpenBSD: perf.html,v 1.24 2010/05/19 13:25:16 sthen Exp $

[Previous: Performance] [Contents] [Next: Authpf: User Shell for Authenticating Gateways]

PF: Issues with FTP

Table of Contents

FTP Modes

FTP is a protocol that dates back to when the Internet was a small, friendly collection of computers and everyone knew everyone else. At that time the need for filtering or tight security wasn't necessary. FTP wasn't designed for filtering, for passing through firewalls, or for working with NAT.

You can use FTP in one of two ways: passive or active. Generally, the choice of active or passive is made to determine who has the problem with firewalling. Realistically, you will have to support both to have happy users.

With active FTP, when a user connects to a remote FTP server and requests information or a file, the FTP server makes a new connection back to the client to transfer the requested data. This is called the *data connection*. To start, the FTP client chooses a random port to receive the data connection on. The client sends the port number it chose to the FTP server and then listens for an incoming connection on that port. The FTP server then initiates a connection to the client's address at the chosen port and transfers the data. This is a problem for users attempting to gain access to FTP servers from behind a NAT gateway. Because of how NAT works, the FTP server initiates the data connection by connecting to the external address of the NAT gateway on the chosen port. The NAT machine will receive this, but because it has no mapping for the packet in its state table, it will drop the packet and won't deliver it to the client.

With passive mode FTP (the default mode with OpenBSD's ftp(1) client), the client requests that the server pick a random port to listen on for the data connection. The server informs the client of the port it has chosen, and the client connects to this port to transfer the data. Unfortunately, this is not always possible or desirable because of the possibility of a firewall in front of the FTP server blocking the incoming data connection. OpenBSD's ftp(1) uses passive mode by default; to force active mode FTP, use the -A flag to ftp, or set passive mode to "off" by issuing the command "`passive off`" at the "`ftp>`" prompt.

FTP Client Behind the Firewall

As indicated earlier, FTP does not go through NAT and firewalls very well.

Packet Filter provides a solution for this situation by redirecting FTP traffic through an FTP proxy server. This process acts to "guide" your FTP traffic through the NAT gateway/firewall, by actively adding needed rules to PF system and removing them when done, by means of the PF anchors system. The FTP proxy used by PF is ftp-proxy(8).

To activate it, put something like this early in the rules section of `pf.conf`:

```
pass in quick on $int_if proto tcp to port 21 rdr-to
127.0.0.1 port 8021
```

This redirects FTP from your clients to the ftp-proxy(8) program, which is listening on your machine to port 8021.

You also need an anchor in the rules section:

```
anchor "ftp-proxy/*"
```

Hopefully it is apparent the proxy server has to be started and running on the OpenBSD box. This is done by inserting the following line in `/etc/rc.conf.local`:

```
ftpproxy_flags=""
```

The ftp-proxy program can be started as root to activate it without a reboot.

ftp-proxy listens on port 8021, the same port the above `rdr-to` statement is sending FTP traffic to.

To support active mode connections from certain (fussy) clients, you may need the '-r' switch on ftp-proxy(8).

PF "Self-Protecting" an FTP Server

In this case, PF is running on the FTP server itself rather than a dedicated firewall computer. When servicing a passive FTP connection, FTP will use a randomly chosen, high TCP port for incoming data. By default, OpenBSD's native FTP server ftpd(8) uses the range 49152 to 65535. Obviously, these must be passed through the filter rules, along with port 21 (the FTP control port):

```
pass in on $ext_if proto tcp to port 21
pass in on $ext_if proto tcp to port > 49151
```

Note that if you desire, you can tighten up that range of ports considerably. In the case of the OpenBSD ftpd(8) program, that is done using the sysctl(8) variables `net.inet.ip.porthifirst` and `net.inet.ip.porthilast`.

FTP Server Protected by an External PF Firewall Running NAT

In this case, the firewall must redirect traffic to the FTP server in addition to not blocking the required ports. In order to accomplish this, we turn again to ftp-proxy(8).

ftp-proxy(8) can be run in a mode that causes it to forward all FTP connections to a specific FTP server. Basically we'll setup the proxy to listen on port 21 of the firewall and forward all connections to the back-end server.

Edit `/etc/rc.conf.local` and add the following:

```
ftpproxy_flags="-R 10.10.10.1 -p 21 -b 192.168.0.1"
```

Here 10.10.10.1 is the IP address of the actual FTP server, 21 is the port we want ftp-proxy(8) to listen on, and 192.168.0.1 is the address on the firewall that we want the proxy to bind to.

Now for the pf.conf rules:

```
ext_ip = "192.168.0.1"
ftp_ip = "10.10.10.1"

match out on $ext_if inet from $int_if nat-to ($ext_if)

anchor "ftp-proxy/*"
pass in on $ext_if inet proto tcp to $ext_ip port 21
pass out on $int_if inet proto tcp to $ftp_ip port 21 user
```

```
    proxy
```

Here we allow the connection inbound to port 21 on the external interface as well as the corresponding outbound connection to the FTP server. The "user proxy" addition to the outbound rule ensures that only connections initiated by ftp-proxy(8) are permitted.

Note that if you want to run ftp-proxy(8) to protect an FTP server as well as allow clients to FTP out from behind the firewall that two instances of ftp-proxy will be required.

More Information on FTP

More information on filtering FTP and how FTP works in general can be found in this whitepaper:

- FTP Reviewed

Proxying TFTP

Trivial File Transfer Protocol (TFTP) suffers from some of the same limitations as FTP does when it comes to passing through a firewall. Luckily, PF has a helper proxy for TFTP called tftp-proxy(8).

tftp-proxy(8) is setup in much the same way as ftp-proxy(8) was in the FTP Client Behind the Firewall section above.

```
match out on $ext_if from $int_if nat-to ($ext_if)
anchor "tftp-proxy/*"
pass in quick on $int_if proto udp from $int_if to port
tftp \
    rdr-to 127.0.0.1 port 6969

anchor "tftp-proxy/*"
```

The rules above allow TFTP outbound from the internal network to TFTP servers on the external network.

The last step is to enable tftp-proxy in inetd.conf(5) so that it listens on the same port that the `rdr-to` rule specified above, in this case 6969.

```
127.0.0.1:6969 dgram udp wait root /usr/libexec/tftp-proxy
tftp-proxy
```

Unlike ftp-proxy(8), tftp-proxy(8) is spawned from inetd.

[Previous: Performance] [Contents] [Next: Authpf: User Shell for Authenticating Gateways]

---

www@openbsd.org

$OpenBSD: ftp.html,v 1.31 2010/05/24 12:13:41 sthen Exp $

[Previous: Issues with FTP] [Contents] [Next: Firewall Redundancy with CARP and pfsync]

PF: Authpf: User Shell for Authenticating Gateways

Table of Contents

Introduction

Authpf(8) is a user shell for authenticating gateways. An authenticating gateway is just like a regular network gateway (a.k.a. a router) except that users must first authenticate themselves to the gateway before it will allow traffic to pass through it. When a user's shell is set to `/usr/sbin/authpf` (i.e., instead of setting a user's shell to ksh(1), csh(1), etc) and the user logs in using SSH, authpf will make the necessary changes to the active pf(4) ruleset so that the user's traffic is passed through the filter and/or translated using Network Address Translation or redirection. Once the user logs out or their session is disconnected, authpf will remove any rules loaded for the user and kill any stateful connections the user has open. Because of this, the ability of the user to pass traffic through the gateway only exists while the user keeps their SSH session open.

Authpf loads a user's rules into a unique anchor point. The anchor is named by combining the user's UNIX username and the authpf process-id into the format "`username(PID)`". Each users' anchor is stored within the `authpf` anchor which is in turn anchored to the main ruleset. The "fully qualified anchor path" then becomes:

```
main_ruleset/authpf/username(PID)
```

The rules that authpf loads can be configured on a per-user or global basis.

Example uses of authpf include:

- Requiring users to authenticate before allowing Internet access.
- Granting certain users -- such as administrators -- access to restricted parts of the network.
- Allowing only known users to access the rest of the network or Internet from a wireless network segment.
- Allowing workers from home, on the road, etc., access to resources on the company network. Users outside the office can not only open access to the company network, but can also be redirected to particular resources (e.g., their own desktop) based on the username they authenticate with.
- In a setting such as a library or other place with public Internet terminals, PF may be configured to allow limited Internet access to guest users. Authpf can then be used to provide registered users with complete access.

Authpf logs the username and IP address of each user who authenticates successfully as well as the start and end times of their login session via syslogd(8). By using this information, an administrator can determine who was logged in when and also make users accountable for their network traffic.

Configuration

The basic steps needed to configure authpf are outlined here. For a complete description of authpf configuration, please refer to the authpf man page.

Enabling Authpf

Authpf will not run if the config file `/etc/authpf/authpf.conf` is not present. The file may be empty (zero size), but unless it is present authpf will exit immediately after a user authenticates successfully.

The following configuration directives can be placed in `authpf.conf`:

- `anchor=`*`name`* - Use the specified anchor name instead of "authpf".
- `table=`*`name`* - Use the specified table name instead of "authpf_users".

Linking Authpf into the Main Ruleset

Authpf is linked into the main ruleset by using an `anchor` rule:

```
anchor "authpf/*"
```

Wherever the `anchor` rule is placed within the ruleset is where PF will branch off from the main ruleset to evaluate the authpf rules.

Configuring Loaded Rules

Authpf loads its rules from one of two files:

- /etc/authpf/users/$USER/authpf.rules
- /etc/authpf/authpf.rules

The first file contains rules that are only loaded when the user $USER (which is replaced with the user's username) logs in. The per-user rule configuration is used when a specific user -- such as an administrator -- requires a set of rules that is different than the default set. The second file contains the default rules which are loaded for any user that doesn't have their own authpf.rules file. If the user-specific file exists, it will override the default file. At least one of the files must exist or authpf will not run.

Rules have the same syntax as in any other PF ruleset with one exception: Authpf allows for the use of two predefined macros:

- $user_ip - the IP address of the logged in user
- $user_id - the username of the logged in user

It's recommended practice to use the $user_ip macro to only permit traffic through the gateway from the authenticated user's computer.

In addition to the $user_ip macro, authpf will make use of the authpf_users table (if it exists) for storing the IP addresses of all authenticated users. Be sure to define the table before using it:

```
table <authpf_users> persist
pass in on $ext_if proto tcp from <authpf_users> \
    to port smtp
```

This table should only be used in rules that are meant to apply to all authenticated users.

Access Control Lists

Users can be prevented from using authpf by creating a file in the /etc/authpf/banned/ directory and naming it after the username that is to be denied access. The contents of the file will be displayed to the user before authpf disconnects them. This provides a handy way to notify the user of why they're disallowed access and who to contact to have their access restored.

Conversely, it's also possible to allow only specific users access by placing usernames in the /etc/authpf/authpf.allow file. If the /etc/authpf/authpf.allow file does not exist or "*" is entered into the file, then authpf will permit access to any user who successfully logs in via SSH as long as they are not explicitly banned.

If authpf is unable to determine if a username is allowed or denied, it will print a brief message and then disconnect the user. An entry in /etc/authpf/banned/ always overrides an entry in /etc/authpf/authpf.allow.

Displaying a Login Message

Whenever a user successfully authenticates to authpf, a greeting is printed that indicates that the user is authenticated.

```
Hello charlie. You are authenticated from host "64.59.56.140"
```

This message can be supplemented by putting a custom message in `/etc/authpf/authpf.message`. The contents of this file will be displayed after the default welcome message.

Assigning Authpf as a User's Shell

In order for authpf to work it must be assigned as the user's login shell. When the user successfully authenticates to sshd(8), authpf will be executed as the user's shell. It will then check if the user is allowed to use authpf, load the rules from the appropriate file, etc.

There are a couple ways of assigning authpf as a user's shell:

1. Manually for each user using chsh(1), vipw(8), useradd(8), usermod(8), etc.
2. By assigning users to a login class and changing the class's `shell` option in `/etc/login.conf`.

Creating an authpf Login Class

When using authpf on a system that has regular user accounts and authpf user accounts, it can be beneficial to use a separate login class for the authpf users. This allows for certain changes to those accounts to be made on a global basis and also allows different policies to be placed on regular accounts and authpf accounts. Some examples of what policies can be set:

- **shell** - Specify a user's login shell. This can be used to force a user's shell to `authpf` regardless of the entry in the passwd(5) database.
- **welcome** - Specify which motd(5) file to display when a user logs in. This is useful for displaying messages that are relevant only to authpf users.

Login classes are created in the login.conf(5) file. OpenBSD comes with an authpf login class defined as:

```
authpf:\
    :welcome=/etc/motd.authpf:\
    :shell=/usr/sbin/authpf:\
    :tc=default:
```

Users are assigned to a login class by editing the `class` field of the user's passwd(5) database entry. One way to do this is with the chsh(1) command.

Seeing Who is Logged In

Once a user has successfully logged in and authpf has adjusted the PF rules, authpf changes its process title to indicate the username and IP address of the logged in user:

```
    # ps -ax | grep authpf
    23664 p0  Is+     0:00.11 -authpf: charlie@192.168.1.3 (authpf)
```

Here the user `charlie` is logged in from the machine 192.168.1.3. By sending a SIGTERM signal to the authpf process, the user can be forcefully logged out. Authpf will also remove any rules loaded for the user and kill any stateful connections the user has open.

```
    # kill -TERM 23664
```

Example

Authpf is being used on an OpenBSD gateway to authenticate users on a wireless network which is part of a larger campus network. Once a user has authenticated, assuming they're not on the banned list, they will be permitted to SSH out and to browse the web (including secure web sites) in addition to accessing either of the campus DNS servers.

The `/etc/authpf/authpf.rules` file contains the following rules:

```
wifi_if = "wi0"

pass in quick on $wifi_if proto tcp from $user_ip to port { ssh, http, \
   https }
```

The administrative user `charlie` needs to be able to access the campus SMTP and POP3 servers in addition to surfing the web and using SSH. The following rules are setup in `/etc/authpf/users/charlie/authpf.rules`:

```
wifi_if = "wi0"
smtp_server = "10.0.1.50"
pop3_server = "10.0.1.51"

pass in quick on $wifi_if \
   proto tcp from $user_ip to $smtp_server port smtp
pass in quick on $wifi_if \
   proto tcp from $user_ip to $pop3_server port pop3
pass in quick on $wifi_if \
   proto tcp from $user_ip to port { ssh, http, https }
```

The main ruleset -- located in `/etc/pf.conf` -- is setup as follows:

```
# macros
wifi_if = "wi0"
ext_if  = "fxp0"
dns_servers = "{ 10.0.1.56, 10.0.2.56 }"

table <authpf_users> persist

# filter
block drop all

pass out quick on $ext_if inet proto { tcp, udp, icmp } \
   from { $wifi_if:network, $ext_if }

pass in quick on $wifi_if inet proto tcp \
   from $wifi_if:network to $wifi_if port ssh

pass in quick on $wifi_if inet proto { tcp, udp } \
   from <authpf_users> to $dns_servers port domain

anchor "authpf/*" in on $wifi_if
```

The ruleset is very simple and does the following:

- Block everything (default deny).
- Pass outgoing TCP, UDP, and ICMP traffic on the external interface from the wireless network and from the gateway itself.
- Pass incoming SSH traffic from the wireless network destined for the gateway itself. This rule is necessary to permit users to log in.
- Pass incoming DNS requests from all authenticated authpf users to the campus DNS servers.
- Create the anchor point "authpf" for incoming traffic on the wireless interface.

The idea behind the main ruleset is to block everything and allow the least amount of traffic through as possible. Traffic is free to flow out on the external interface but is blocked from entering the wireless interface by the default deny policy. Once a user authenticates, their traffic is permitted to pass in on the wireless interface and to then flow through the gateway into the rest of the network. The `quick` keyword is used throughout so that PF doesn't have to evaluate each named ruleset when a new connection passes through the gateway.

[Previous: Issues with FTP] [Contents] [Next: Firewall Redundancy with CARP and pfsync]

---

www@openbsd.org

$OpenBSD: authpf.html,v 1.27 2010/05/19 13:25:16 sthen Exp $

OpenBSD

[Previous: Authpf: User Shell for Authenticating Gateways] [Contents] [Next: Firewall for Home or Small Office]

PF: Firewall Redundancy with CARP and pfsync

Table of Contents

Introduction to CARP

CARP is the Common Address Redundancy Protocol. Its primary purpose is to allow multiple hosts on the same network segment to share an IP address. CARP is a secure, free alternative to the Virtual Router Redundancy Protocol (VRRP) and the Hot Standby Router Protocol (HSRP).

CARP works by allowing a group of hosts on the same network segment to share an IP address. This group of hosts is referred to as a "redundancy group". The redundancy group is assigned an IP address that is shared amongst the group members. Within the group, one host is designated the "master" and the rest as "backups". The master host is the one that currently "holds" the shared IP; it responds to any traffic or ARP requests directed towards it. Each host may belong to more than one redundancy group at a time.

One common use for CARP is to create a group of redundant firewalls. The virtual IP that is assigned to the redundancy group is configured on client machines as the default gateway. In the event that the master firewall suffers a failure or is taken offline, the IP will move to one of the backup firewalls and service will continue unaffected.

CARP supports IPv4 and IPv6.

CARP Operation

The master host in the group sends regular advertisements to the local network so that the backup hosts know it's still alive. If the backup hosts don't hear an advertisement from the master for a set period of time, then one of them will take over the duties of master (whichever backup host has the lowest configured `advbase` and `advskew` values).

It's possible for multiple CARP groups to exist on the same network segment. CARP advertisements contain the Virtual Host ID which allows group members to identify which redundancy group the advertisement belongs to.

In order to prevent a malicious user on the network segment from spoofing CARP advertisements, each group can be configured with a password. Each CARP packet sent to the group is then protected by an SHA1 HMAC.

Since CARP is its own protocol it should have an explicit pass rule in filter rulesets:

```
pass out on $carp_dev proto carp keep state
```

`$carp_dev` should be the physical interface that CARP is communicating over.

Configuring CARP

Each redundancy group is represented by a carp(4) virtual network interface. As such, CARP is configured using ifconfig(8).

```
ifconfig carpN create

ifconfig carpN vhid vhid [pass password] [carpdev carpdev] \
   [advbase advbase] [advskew advskew] [state state] [group|-group
group] \
   ipaddress netmask mask
```

carpN

The name of the carp(4) virtual interface where *N* is an integer that represents the interface's number (e.g. carp10).

vhid

The Virtual Host ID. This is a unique number that is used to identify the redundancy group to other nodes on the network. Acceptable values are from 1 to 255.

password

The authentication password to use when talking to other CARP-enabled hosts in this redundancy group. This must be the same on all members of the group.

carpdev

This optional parameter specifies the physical network interface that belongs to this redundancy group. By default, CARP will try to determine which interface to use by looking for a physical interface that is in the same subnet as the *ipaddress* and *mask* combination given to the carp(4) interface.

`advbase`

This optional parameter specifies how often, in seconds, to advertise that we're a member of the redundancy group. The default is 1 second. Acceptable values are from 1 to 255.

`advskew`

This optional parameter specifies how much to skew the *`advbase`* when sending CARP advertisements. By manipulating *`advskew`*, the master CARP host can be chosen. The higher the number, the *less* preferred the host will be when choosing a master. The default is 0. Acceptable values are from 0 to 254.

`state`

Force a carp(4) interface into a certain state. Valid states are `init`, `backup`, and `master`.

`group, -group`

Add or remove a carp(4) interface to a certain interface group. By default all carp(4) interfaces are added to the `carp` group. Each group has a `carpdemote` counter affecting all carp(4) interfaces belonging to that group. As described below, it can be useful to group certain interfaces together for failover purposes.

`ipaddress`

This is the shared IP address assigned to the redundancy group. This address does not have to be in the same subnet as the IP address on the physical interface (if present). This address needs to be the same on all hosts in the group, however.

`mask`

The subnet mask of the shared IP.

Further CARP behavior can be controlled via sysctl(8).

`net.inet.carp.allow`

Accept incoming CARP packets or not. Default is 1 (yes).

`net.inet.carp.preempt`

Allow hosts within a redundancy group that have a better `advbase` and `advskew` to preempt the master. In addition, this option also enables failing over a group of interfaces together in the event that one interface goes down. If one physical CARP-enabled interface goes down, CARP will increase the demotion counter, `carpdemote`, by 1 on interface groups that the carp(4) interface is a member of, in effect causing all group members to fail-over together. `net.inet.carp.preempt` is 0 (disabled) by default.

`net.inet.carp.log`

Log state changes, bad packets and other errors. May be between 0 and 7, corresponding with syslog(3) priorities. The default is 2 (state changes only).

CARP Example

Here is an example CARP configuration:

```
# sysctl -w net.inet.carp.allow=1
# ifconfig carp1 create
# ifconfig carp1 vhid 1 pass mekmitasdigoat carpdev em0 \
    advskew 100 10.0.0.1 netmask 255.255.255.0
```

This sets up the following:

- Enables receipt of CARP packets (this is the default setting).
- Creates a carp(4) interface, `carp1`.
- Configures `carp1` for virtual host #1, enables a password, sets `em0` as the interface belonging to the group, and makes this host a backup due to the `advskew` of `100` (assuming of course that the master is set up with an `advskew` less than 100). The shared IP assigned to this group is 10.0.0.1/255.255.255.0.

Running `ifconfig` on `carp1` shows the status of the interface.

```
# ifconfig carp1
carp1: flags=8802<UP,BROADCAST,SIMPLEX,MULTICAST> mtu 1500
     carp: BACKUP carpdev em0 vhid 1 advbase 1 advskew 100
     groups: carp
     inet 10.0.0.1 netmask 0xffffff00 broadcast 10.0.0.255
```

Introduction to pfsync

The pfsync(4) network interface exposes certain changes made to the pf(4) state table. By monitoring this device using tcpdump(8), state table changes can be observed in real time. In addition, the pfsync(4) interface can send these state change messages out on the network so that other nodes running PF can merge the changes into their own state tables. Likewise, pfsync(4) can also listen on the network for incoming messages.

pfsync Operation

By default, pfsync(4) does not send or receive state table updates on the network; however, updates can still be monitored using tcpdump(8) or other such tools on the local machine.

When pfsync(4) is set up to send and receive updates on the network, the default behavior is to multicast updates out on the local network. All updates are sent without authentication. Best common practice is either:

1. Connect the two nodes that will be exchanging updates back-to-back using a crossover cable and use that interface as the `syncdev` (see below).
2. Use the ifconfig(8) `syncpeer` option (see below) so that updates are unicast directly to the peer, then configure ipsec(4) between the hosts to secure the pfsync(4) traffic.

When updates are being sent and received on the network, pfsync packets should be passed in the filter ruleset:

```
pass on $sync_if proto pfsync
```

`$sync_if` should be the physical interface that pfsync(4) is communicating over.

Configuring pfsync

Since pfsync(4) is a virtual network interface, it is configured using ifconfig(8).

```
ifconfig pfsyncN syncdev syncdev [syncpeer syncpeer] [defer|-defer]
```

pfsyncN
: The name of the pfsync(4) interface. `pfsync0` exists by default when using the `GENERIC` kernel.

syncdev
: The name of the physical interface used to send pfsync updates out.

syncpeer
: This optional parameter specifies the IP address of a host to exchange pfsync updates with. By default pfsync updates are multicast on the local network. This option overrides that behavior and instead unicasts the update to the specified *syncpeer*.

defer
: If the `defer` flag is used, the initial packet of a new connection passing through the firewall will not be transmitted until either another pfsync(4) system has acknowledged the state table addition, or a timeout has expired. This adds small delays but allows traffic to flow when more than one firewall might actively handle packets ("active/active"), e.g. with certain ospfd(8), bgpd(8) or carp(4) configurations.

pfsync Example

Here is an example pfsync configuration:

```
# ifconfig pfsync0 syncdev em1
```

This enables pfsync on the `em1` interface. Outgoing updates will be multicast on the network allowing any other host running pfsync to receive them.

Combining CARP and pfsync For Failover

By combining the features of CARP and pfsync, a group of two or more firewalls can be used to create a highly-available, fully redundant firewall cluster.

CARP:
: Handles the automatic failover of one firewall to another.

pfsync:
: Synchronizes the state table amongst all the firewalls. In the event of a failover, traffic can flow uninterrupted through the new master firewall.

An example scenario. Two firewalls, `fw1` and `fw2`.

```
    +----| WAN/Internet |----+
    |                        |
 em2|                        |em2
 +-----+                  +-----+
 | fw1 |-em1----------em1-| fw2 |
 +-----+                  +-----+
 em0|                        |em0
    |                        |
 ---+-------Shared LAN-------+---
```

The firewalls are connected back-to-back using a crossover cable on `em1`. Both are connected to the LAN on `em0` and to a WAN/Internet connection on `em2`. IP addresses are as follows:

- fw1 em0: 172.16.0.1
- fw1 em1: 10.10.10.1
- fw1 em2: 192.0.2.1
- fw2 em0: 172.16.0.2
- fw2 em1: 10.10.10.2
- fw2 em2: 192.0.2.2
- LAN shared IP: 172.16.0.100
- WAN/Internet shared IP: 192.0.2.100

The network policy is that `fw1` will be the preferred master.

Configure fw1:

```
! enable preemption and group interface failover
# sysctl -w net.inet.carp.preempt=1

! configure pfsync
# ifconfig em1 10.10.10.1 netmask 255.255.255.0
# ifconfig pfsync0 syncdev em1
# ifconfig pfsync0 up

! configure CARP on the LAN side
# ifconfig carp1 create
# ifconfig carp1 vhid 1 carpdev em0 pass lanpasswd \
     172.16.0.100 netmask 255.255.255.0

! configure CARP on the WAN/Internet side
# ifconfig carp2 create
# ifconfig carp2 vhid 2 carpdev em2 pass netpasswd \
    192.0.2.100 netmask 255.255.255.0
```

Configure fw2:

```
! enable preemption and group interface failover
# sysctl -w net.inet.carp.preempt=1

! configure pfsync
# ifconfig em1 10.10.10.2 netmask 255.255.255.0
# ifconfig pfsync0 syncdev em1
# ifconfig pfsync0 up

! configure CARP on the LAN side
# ifconfig carp1 create
# ifconfig carp1 vhid 1 carpdev em0 pass lanpasswd \
     advskew 128 172.16.0.100 netmask 255.255.255.0

! configure CARP on the WAN/Internet side
# ifconfig carp2 create
# ifconfig carp2 vhid 2 carpdev em2 pass netpasswd \
    advskew 128 192.0.2.100 netmask 255.255.255.0
```

Operational Issues

Some common operational issues encountered with CARP/pfsync.

Configuring CARP and pfsync During Boot

Since carp(4) and pfsync(4) are both types of network interfaces, they can be configured at boot by creating a hostname.if(5) file. The netstart startup script will take care of creating the interface and configuring it.

Examples:

```
/etc/hostname.carp1
        inet 172.16.0.100 255.255.255.0 172.16.0.255 vhid 1 carpdev em0 \
           pass lanpasswd

/etc/hostname.pfsync0
        up syncdev em1
```

Forcing Failover of the Master

There can be times when it's necessary to failover or demote the master node on purpose. Examples include taking the master node down for maintenance or when troubleshooting a problem. The objective here is to gracefully fail over traffic to one of the backup hosts so that users do not notice any impact.

To failover a particular CARP group, shut down the carp(4) interface on the master node. This will cause the master to advertise itself with an "infinite" `advbase` and `advskew`. The backup host(s) will see this and immediately take over the role of master.

```
# ifconfig carp1 down
```

An alternative is to increase the `advskew` to a value that's higher than the `advskew` on the backup host(s). This will cause a failover but still allow the master to participate in the CARP group.

Another method of failover is to tweak the CARP demotion counter. The demotion counter is a measure of how "ready" a host is to become master of a CARP group. For example, while a host is in the middle of booting up it's a bad idea for it to become the CARP master until all interfaces have been configured, all network daemons have been started, etc. Hosts advertising a high demotion value will be less preferred as the master.

A demotion counter is stored in each interface group that the CARP interface belongs to. By default, all CARP interfaces are members of the "carp" interface group. The current value of a demotion counter can be viewed using ifconfig(8):

```
# ifconfig -g carp
carp: carp demote count 0
```

In this example the counter associated with the "carp" interface group is shown. When a CARP host advertises itself on the network, it takes the sum of the demotion counters for each interface group the carp(4) interface belongs to and advertises that value as its demotion value.

Now assume the following example. Two firewalls running CARP with the following CARP interfaces:

- carp1 -- Accounting Department
- carp2 -- Regular Employees
- carp3 -- Internet
- carp4 -- DMZ

The objective is to failover just the carp1 and carp2 groups to the secondary firewall.

First, assign each to a new interface group, in this case named "internal":

```
# ifconfig carp1 group internal
# ifconfig carp2 group internal
# ifconfig internal
carp1: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> mtu 1500
     carp: MASTER carpdev em0 vhid 1 advbase 1 advskew 100
     groups: carp internal
     inet 10.0.0.1 netmask 0xffffff00 broadcast 10.0.0.255
carp2: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> mtu 1500
     carp: MASTER carpdev em1 vhid 2 advbase 1 advskew 100
     groups: carp internal
     inet 10.0.1.1 netmask 0xffffff00 broadcast 10.0.1.255
```

Now increase the demotion counter for the "internal" group using ifconfig(8):

```
# ifconfig -g internal
internal: carp demote count 0
# ifconfig -g internal carpdemote 50
# ifconfig -g internal
internal: carp demote count 50
```

The firewall will now gracefully failover on the carp1 and carp2 groups to the other firewall in the cluster while still remaining the master on carp3 and carp4. If the other firewall started advertising itself with a demotion value higher than 50, or if the other firewall stopped advertising altogether, then this firewall would again take over mastership on carp1 and carp2.

To fail back to the primary firewall, reverse the changes:

```
# ifconfig -g internal -carpdemote 50
# ifconfig -g internal
internal: carp demote count 0
```

Network daemons such as OpenBGPD and sasyncd(8) make use of the demotion counter to ensure that the firewall does not become master until BGP sessions become established and IPsec SAs are synchronized.

Ruleset Tips

Filter the physical interface. As far as PF is concerned, network traffic comes from the physical interface, not the CARP virtual interface (i.e., `carp0`). So, write your rule sets accordingly. Don't forget that an interface name in a PF rule can be either the name of a physical interface or an address associated with that interface. For example, this rule could be correct:

```
pass in on fxp0 inet proto tcp from any to carp0 port 22
```

but replacing the `fxp0` with `carp0` would not work as you desire.

DON'T forget to pass `proto carp` and `proto pfsync`!

Other References

Please see these other sources for more information:

- carp(4)
- pfsync(4)
- ifconfig(8)
- hostname.if(5)
- pf.conf(5)

- ifstated(8)
- ifstated.conf(5)

[Previous: Authpf: User Shell for Authenticating Gateways] [Contents] [Next: Firewall for Home or Small Office]

---

www@openbsd.org

$OpenBSD: carp.html,v 1.26 2010/05/19 13:25:16 sthen Exp $

[Previous: Firewall Redundancy with CARP and pfsync] [Contents]

PF: Example: Firewall for Home or Small Office

Table of Contents

The Scenario

In this example, PF is running on an OpenBSD machine acting as a firewall and NAT gateway for a small network in a home or office. The overall objective is to provide Internet access to the network and to allow limited access to the firewall machine from the Internet, and expose an internal web server to the external Internet. This document will go through a complete ruleset that does just that.

The Network

The network is setup like this:

```
  [ COMP1 ]    [ COMP3 ]
      |            |
   ---+------+-----+------- xl0 [ OpenBSD ] fxp0 -------- ( Internet )
             |
         [ COMP2 ]
```

There are a number of computers on the internal network; the diagram shows three but the actual number is irrelevant. These computers are regular workstations used for web surfing, email, chatting, etc., except for COMP3 which is also running a small web server. The internal network is using the 192.168.0.0 / 255.255.255.0 network block.

The OpenBSD firewall is a Celeron 300 with two network cards: a 3com 3c905B (`xl0`) and an Intel EtherExpress Pro/100 (`fxp0`). The firewall has a cable connection to the Internet and is using NAT to share this connection with the internal network. The IP address on the external interface is dynamically assigned by the Internet Service Provider.

The Objective

The objectives are:

- Provide unrestricted Internet access to each internal computer.
- Use a "default deny" filter ruleset.
- Allow the following incoming traffic to the firewall from the Internet:
    - SSH (TCP port 22): this will be used for external maintenance of the firewall machine.
    - Auth/Ident (TCP port 113): used by some services such as SMTP and IRC.
    - ICMP Echo Requests: the ICMP packet type used by ping(8).
- Redirect TCP port 80 connection attempts (which are attempts to access a web server) to computer COMP3. Also, permit TCP port 80 traffic destined for COMP3 through the firewall.
- Log filter statistics on the external interface.
- By default, reply with a TCP RST or ICMP Unreachable for blocked packets.
- Make the ruleset as simple and easy to maintain as possible.

Preparation

This document assumes that the OpenBSD host has been properly configured to act as a router, including verifying IP networking setup, Internet connectivity, and setting the sysctl(3) variables `net.inet.ip.forwarding` and/or `net.inet6.ip6.forwarding` to "`1`". You must also have enabled PF using pfctl(8) or by setting the appropriate variable in `/etc/rc.conf.local`. PF is enabled by default on OpenBSD 4.6 and newer releases.

The Ruleset

The following will step through a ruleset that will accomplish the above goals.

Macros

The following macros are defined to make maintenance and reading of the ruleset easier:

```
int_if="xl0"
```

```
tcp_services="{ 22, 113 }"
icmp_types="echoreq"

comp3="192.168.0.3"
```

The first line defines the internal network interface that filtering will happen on. By defining them here, if we have to move this system to another machine with different hardware, we can change only those two lines, and the rest of the rule set will be still usable. (For this example, the external interface will be handled by using the `egress` interface group. This is automatically set on any interface holding a default route, in this case, fxp0). The second and third lines list the TCP port numbers of the services that will be opened up to the Internet (SSH and ident/auth) and the ICMP packet types that will be accepted at the firewall machine. Finally, the last line defines the IP address of COMP3.

Note: If the Internet connection required PPPoE, then filtering and NAT would have to take place on the `pppoe0` interface and *not* on `fxp0`.

Options

The following two options will set the default response for `block` filter rules and turn statistics logging "on" for the external interface:

```
set block-policy return
set loginterface fxp0
```

Every Unix system has a "loopback" interface. It's a virtual network interface that is used by applications to talk to each other inside the system. On OpenBSD, the loopback interface is lo(4). It is considered best practice to disable all filtering on loopback interfaces. Using set skip will accomplish this.

```
set skip on lo
```

Note that we are skipping for all `lo` interfaces, this way, should we later add additional loopback interfaces, we won't have to worry about altering this portion of our existing rules file.

Firewall Rules

We will start with rules to support the use of ftp-proxy(8) so that FTP clients on the local network can connect to FTP servers on the Internet. This works by dynamically inserting rules when an ftp connection is made. This is done using an anchor:

```
anchor "ftp-proxy/*"
```

Now we will add the rule needed to redirect FTP connections so they are seen by ftp-proxy(8):

```
pass in quick on $int_if inet proto tcp to any port ftp \
    rdr-to 127.0.0.1 port 8021
```

This rule will intercept FTP connections to port 21 and redirect them to an ftp-proxy(8) instance running on port 8021 and, through the use of the `quick` keyword, matching packets will not be further checked against the rest of the ruleset. If users regularly connect to FTP servers on other ports, then a list should be used to specify the destination port, for example: `to any port { 21, 2121 }`.

Note that both the anchor and the ftp-proxy(8) redirect rule need to be located before any `match` rules for NAT or the ftp-proxy(8) will not work as expected.

Now we move on to some `match` rules. By itself, a `match` rule doesn't determine whether or not a packet is allowed to pass. Instead, packets matching this rule will have the parameters remembered; they will then be used in any `pass` rules which handle these packets.

This is powerful: parameters such as NAT or queueing can be applied to certain classes of packet, and then access permissions can be defined separately.

To perform NAT for the entire internal network the following `match` rule is used:

```
    match out on egress inet from !(egress:network) to any nat-to
    (egress:0)
```

In this case, the "`!(egress:network)`" could easily be replaced by a "`$int_if:network`", but if you added multiple internal interfaces, you would have to add additional NAT rules, whereas with this structure, NAT will be handled on all protected interfaces.

Since the IP address on the external interface is assigned dynamically, parenthesis are placed around the translation interface so that PF will notice when the address changes. The :0 suffix is used so that, if the external interface has multiple addresses, only the first address is used for translation.

Lastly, the protocol family `inet` (IPv4) is specified. This avoids translating any `inet6` (IPv6) packets which may be received.

Now the rules to control access permissions. Start with the default deny:

```
    block in log
```

At this point all traffic attempting to come into an interface will be blocked, even that from the internal network. These packets will also be logged. Later rules will open up the firewall as per the objectives above as well as open up any necessary virtual interfaces.

Keep in mind, pf can block traffic coming into or leaving out of an interface. It can simplify your life if you choose to filter traffic in one direction, rather than trying to keep it straight when filtering some things in, and some things out. In our case, we'll opt to filter the inbound traffic, but once the traffic is permitted into an interface, we won't try to obstruct it leaving, so we will do the following:

```
    pass out quick
```

By using `quick`, outbound packets can avoid being checked against the following rules, improving performance.

It is good to use the spoofed address protection:

```
    antispoof quick for { lo $int_if }
```

Now open the ports used by those network services that will be available to the Internet. First, the traffic that is destined to the firewall itself:

```
    pass in on egress inet proto tcp from any to (egress) \
        port $tcp_services
```

Specifying the network ports in the macro `$tcp_services` makes it simple to open additional services to the Internet by simply editing the macro and reloading the ruleset. UDP services can also be opened up by creating a `$udp_services` macro and adding a filter rule, similar to the one above, that specifies `proto udp`.

The next rule catches any attempts by someone on the Internet to connect to TCP port 80 on the firewall. Legitimate attempts to access this port will be from users trying to access the network's web server. These connection attempts need to be redirected to COMP3:

```
    pass in on egress inet proto tcp to (egress) port 80 \
        rdr-to $comp3 synproxy state
```

For an added bit of safety, we'll make use of the TCP SYN Proxy to further protect the web server.

ICMP traffic needs to be passed:

```
    pass in inet proto icmp all icmp-type $icmp_types
```

Similar to the `$tcp_services` macro, the `$icmp_types` macro can easily be edited to change the types of ICMP packets that will be allowed to reach the firewall. Note that this rule applies to all network interfaces.

Now traffic must be passed to and from the internal network. We'll assume that the users on the internal network know what they are doing and aren't going to be causing trouble. This is not necessarily a valid assumption; a much more restrictive ruleset would be appropriate for many environments.

```
    pass in on $int_if
```

TCP, UDP, and ICMP traffic is permitted to exit the firewall towards the Internet due to the earlier "`pass out`" line. State information is kept so that the returning packets will be passed back in through the firewall.

The Complete Ruleset

```
# macros

int_if="xl0"

tcp_services="{ 22, 113 }"
icmp_types="echoreq"

comp3="192.168.0.3"

# options

set block-policy return
set loginterface fxp0
set skip on lo

# FTP Proxy rules

anchor "ftp-proxy/*"

pass in quick on $int_if inet proto tcp to any port ftp \
    rdr-to 127.0.0.1 port 8021

# match rules

match out on egress inet from !(egress) to any nat-to (egress:0)

# filter rules

block in log
pass out quick

antispoof quick for { lo $int_if }

pass in on egress inet proto tcp from any to (egress) \
    port $tcp_services

pass in on egress inet proto tcp to (egress) port 80 \
    rdr-to $comp3 synproxy state

pass in inet proto icmp all icmp-type $icmp_types

pass in on $int_if
```

[Previous: Firewall Redundancy with CARP and pfsync] [Contents]

www@openbsd.org

$OpenBSD: example1.html,v 1.43 2010/07/19 23:43:08 sthen Exp $