OpenBSD

Languages: [en] [es] [id] [ja] [pl]

[up to FAQ] [Next: Configuration]

PF: The OpenBSD Packet Filter

Table of Contents

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 Daniel and the rest of the OpenBSD team.

This set of documents is intended as a general introduction to the PF system as run on OpenBSD. It is intended to be used as a supplement to the man pages, not as a replacement for them. This document does not cover all of PF's features and may not be as up to date as the man pages are.

As with the rest of the FAQ, this document is focused on users of OpenBSD 3.3. As PF is always growing and developing, there are changes and enhancements between the 3.3-release version and the version in OpenBSD-current. The reader is advised to see the man pages for the version of OpenBSD they are currently working with.

[up to FAQ] [Next: Configuration]


www@openbsd.org

$OpenBSD: index.html,v 1.8 2003/09/07 15:55:23 nick Exp $

[Contents] [Next: Lists and Macros]

PF: Configuration

Table of Contents

Activation

To activate PF and have it read its configuration file at boot, edit the `/etc/rc.conf` file so that the PF line reads:

```
pf=YES
```

Reboot your system to have it take effect.

You can also 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 rulset 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 seven 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.
- **Scrub:** Reprocessing packets to normalize and defragment them.
- **Queueing:** Provides bandwidth control and packet prioritization.
- **Translation:** Controls Network Address Translation and packet redirection.
- **Filter Rules:** Allows the selective filtering or blocking of packets as they pass through any of the interfaces.

With the exception of macros and tables, each section should appear in this order in the configuration file, though not all sections have to exist for any particular application.

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     loads the pf.conf file
# pfctl -nf /etc/pf.conf    parse the file, but don't load it
# pfctl -Nf /etc/pf.conf    Load only the NAT rules from the file
# pfctl -Rf /etc/pf.conf    Load only the filter rules from the file

# pfctl -sn                 Show the current NAT rules
# pfctl -sr                 Show the current filter rules
# 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) manpage.

[Contents] [Next: Lists and Macros]


www@openbsd.org

$OpenBSD: config.html,v 1.7 2003/09/22 02:29:03 nick Exp $

[Previous: Configuration] [Contents] [Next: Tables]

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 and are not limited to just filter rules:

```
rdr on fxp0 proto tcp from any to any port { 22 80 } -> \
   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.

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: Configuration] [Contents] [Next: Tables]


www@openbsd.org

$OpenBSD: macros.html,v 1.6 2003/09/22 02:29:03 nick Exp $

[Previous: Lists and Macros] [Contents] [Next: Options]

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 source and destination address fields in filter, `scrub`, `nat`, and redirection rules, but they cannot be used as the redirection address in `nat` rules or in the routing options (`route-to`, `reply-to`, `dup-to`) of filter rules. Tables are created either in `/etc/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 < >.

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, 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 -Tadd 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 -Tshow
```

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 -Tdelete 218.70.0.0/16
```

For more information on manipulating tables with `pfctl`, please see `pfctl(8)`.

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 or the `self` keyword in which case all addresses assigned to the interface(s) will be added to the table.

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 all
pass  in on dc0 from <goodguys> to any
```

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: Options]


www@openbsd.org

$OpenBSD: tables.html,v 1.6 2003/09/22 02:29:03 nick Exp $

OpenBSD

[Previous: Tables] [Contents] [Next: Scrub]

PF: Options

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

`set block-policy`

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 blocked UDP packets. All other packets are silently dropped.

Note that individual filter rules can override the default response.

`set limit`

`frags` - maximum number of entries in the memory pool used for packet reassembly (scrub rules). Default is 5000.

`states` - maximum number of entries in the memory pool used for state table entries (filter rules that specify `keep state`). Default is 10000.

`set loginterface` *`int`*

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.

`set optimization`

Optimize PF for one of the following network environments:

- `normal` - suitable for almost all networks. This is the default.
- `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.

`set timeout`

`interval` - seconds between purges of expired states and packet fragments.

`frag` - seconds before an unassembled fragment is expired.

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
```

[Previous: Tables] [Contents] [Next: Scrub]

---

www@openbsd.org

$OpenBSD: options.html,v 1.4 2003/09/22 02:29:03 nick Exp $

[Previous: Options] [Contents] [Next: Queueing]

PF: Scrub

Table of Contents

Introduction

"Scrubbing" is the normalization of packets so there are no ambiguities in interpretation by the ultimate destination of the packet. The scrub directive also reassembles fragmented packets, protecting some operating systems from some forms of attack, and drops TCP packets that have invalid flag combinations. A simple form of the scrub directive:

```
scrub in all
```

This will scrub all incoming packets on all interfaces.

One reason not to scrub on an interface is if one is passing NFS through PF. Some non-OpenBSD platforms send (and expect) strange packets -- fragmented packets with the "do not fragment" bit set, which are (properly) rejected by `scrub`. This can be resolved by use of the `no-df` option. Another reason is some multi-player games have connection problems passing through PF with `scrub` enabled. Other than these somewhat unusual cases, scrubbing all packets is *highly recommended* practice.

The `scrub` directive syntax is very similar to the filtering syntax which makes it easy to selectively scrub certain packets and not others.

More on the principle and concepts of scrubbing here: http://www.icir.org/vern/papers/norm-usenix-sec-01-html/index.html

Options

`Scrub` has the following options:

`no-df`
: Clears the *don't fragment* bit from the IP packet header. Some operating systems are known to generate fragmented packets with the *don't fragment* bit set. This is particularly true with NFS. `Scrub` will drop such packets unless the `no-df` option is specified. Because some operating systems generate *don't fragment* packets with a zero IP identification header field, using `no-df` in conjunction with `random-id` is recommended.

`random-id`

Replaces the IP identification field of outgoing packets with random values to compensate for operating systems that use predictable values. This option only applies to outgoing packets that are not fragmented after the optional packet reassembly.

`min-ttl` *num*
: Enforces a minimum Time To Live (TTL) in IP packet headers.

`max-mss` *num*
: Enforces a maximum Maximum Segment Size (MSS) in TCP packet headers.

`fragment reassemble`
: Buffers incoming packet fragments and reassembles them into a complete packet before passing them to the filter engine. The advantage is that filter rules only have to deal with complete packets and can ignore fragments. The drawback is the increased memory needed to buffer packet fragments. This is the default behavior when no `fragment` option is specified. This is also the only `fragment` option that works with NAT.

`fragment crop`
: Causes duplicate fragments to be dropped and any overlaps to be cropped. Unlike `fragment reassemble`, fragments are not buffered but are passed on as soon as they arrive.

`fragment drop-ovl`
: Similar to `fragment crop` except that all duplicate or overlapping fragments will be dropped as well as any further corresponding fragments.

Example:

```
scrub in on fxp0 all fragment reassemble min-ttl 15 max-mss 1400
```

[Previous: Options] [Contents] [Next: Queueing]


www@openbsd.org

$OpenBSD: scrub.html,v 1.6 2003/09/22 02:29:03 nick Exp $

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

PF: Queueing

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 or a bank -- 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, 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 visit this website.

Priority Queueing

Priority Queueing (PRIQ) assigns multiple queues to a network interface with each queue being given a unique priority level. A queue with a higher priority is *always* processed ahead of a queue with a lower priority.

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 visit this website.

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 the configuration section.

Queueing is configured in `/etc/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 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.
- *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.
- *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 { ssh_login, ssh_bulk }
  queue ssh_login priority 4 cbq(ecn)
  queue ssh_bulk 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 defines 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.

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 from any to any port 22
```

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

```
pass out on fxp0 from any to any port 22 queue ssh
```

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 tagging 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 cbq(default)
queue ftp bandwidth 1.5Mb

pass in on dc0 from any to any port 21 queue ftp
```

Queueing is enabled on fxp0 but the tagging takes place on dc0. If packets matching the `pass` 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.

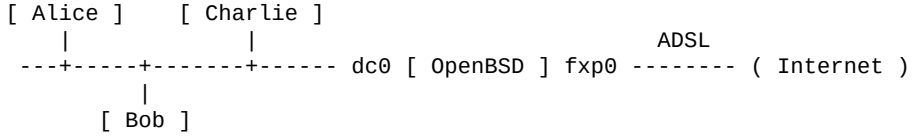
Assigning TCP ACK packets to a higher priority queue is useful on asymmetric connections, that is, connections that have different upload and download bandwidths such as ADSL lines. With an ADSL line, if the upload channel is being maxed out and a download is started, the download will suffer because 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.

When using the `queue` keyword with rules that `keep state` such as:

```
pass in on fxp0 proto tcp from any to any port 22 flags S/SA \
   keep state queue ssh
```

PF will record the queue in the state table entry so that packets traveling back out fxp0 that match the stateful connection will end up in the `ssh` 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.

Example #1: Small, Home Network



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:

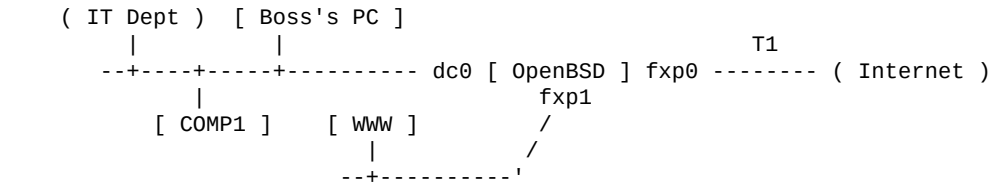
- Reserve 80Kbps of download bandwidth for Bob so he can play his online games without being lagged by Alice or Charlie's downloads. Allow Bob to use more than 80Kbps when it's available.
- Interactive SSH and instant message traffic will have a higher priority than regular traffic.
- DNS queries and 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; `nat`, `rdr`, options, etc., are not shown.

```
# 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    cbq(default)
queue ssh_im_in priority 4
queue dns_in    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) to any flags S/SA \
        keep state queue(std_out, tcp_ack_out)
pass  out on fxp0 inet proto { udp icmp } from (fxp0) to any keep state
pass  out on fxp0 inet proto { tcp udp } from (fxp0) to any port domain \
        keep state queue dns_out
pass  out on fxp0 inet proto tcp from (fxp0) to any port $ssh_ports \
        flags S/SA keep state queue(std_out, ssh_im_out)
pass  out on fxp0 inet proto tcp from (fxp0) to any port $im_ports \
        flags S/SA keep state 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 from any to $local_net
pass  out on dc0 proto { tcp udp } from any port domain to $local_net \
        queue dns_in
```

```
pass  out on dc0 proto tcp from any port $ssh_ports to $local_net \
        queue(std_in, ssh_im_in)
pass  out on dc0 proto tcp from any port $im_ports to $local_net \
        queue ssh_im_in
pass  out on dc0 from any to $bob queue bob_in
```

Example #2: Company Network



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 traffic between the WWW server and the Internet to 500Kbps in each direction.
- No bandwidth limit on traffic between the WWW server and the internal network.
- 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).
- 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
#   www_ext_misc - all non-http traffic from the WWW server
# boss_ext       - traffic coming from the boss's computer

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

# 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.
#   boss_int - traffic to the boss's PC.
# www_int    - traffic from the WWW server in the DMZ.

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

# 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.
#   net_dmz_misc - all non-http traffic. this is also the default queue.

queue internal_dmz      # no special settings needed
queue net_dmz        bandwidth 500Kb { net_dmz_http, net_dmz_misc }
  queue net_dmz_http priority 3 cbq(red)
  queue net_dmz_misc priority 1 cbq(default)


# ... 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 } flags S/SA keep state queue www_ext_misc
pass in on fxp0 proto tcp from any to $wwwserv port 80 \
        flags S/SA keep state queue www_ext_http

# filter rules for fxp0 outbound
pass out on fxp0 from $int_nets to any keep state
pass out on fxp0 from $boss to any keep state queue boss_ext

# filter rules for dc0 inbound
pass in on dc0 from $int_nets to any keep state
pass in on dc0 from $it_net to any queue it_int
pass in on dc0 from $boss to any queue boss_int
pass in on dc0 proto tcp from $int_nets to $wwwserv port { 21, 80, \
        > 49151 } flags S/SA keep state 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 any port 53 \
        keep state

# filter rules for fxp1 outbound
pass out on fxp1 proto tcp from any to $wwwserv port { 21, \
        > 49151 } flags S/SA keep state queue net_dmz_misc
pass out on fxp1 proto tcp from any to $wwwserv port 80 \
```

```
        flags S/SA keep state queue net_dmz_http
pass out on fxp1 proto tcp from $int_nets to $wwwserv port { 80, \
        21, > 49151 } flags S/SA keep state queue internal_dmz
```

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


www@openbsd.org

$OpenBSD: queueing.html,v 1.12 2003/09/22 02:29:03 nick Exp $

[Previous: Queueing] [Contents] [Next: Traffic Redirection]

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.

NAT allows you to take advantage of the reserved address blocks described in RFC 1918. 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.

Enabling NAT

To enable NAT on an OpenBSD gateway, in addition to activating PF, you must also enable IP forwarding:

```
# sysctl -w net.inet.ip.forwarding=1
# sysctl -w 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

The general format for NAT rules in `/etc/pf.conf` looks something like this:

```
nat on extif [af] from src_addr [port src_port] to \
   dst_addr [port dst_port] -> ext_addr
```

extif
: The name of the external network interface.

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).

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 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 of a network interface followed by the `:network` keyword. The resulting CIDR network (e.g. 192.168.0.0/24) will be substituted into the rule when the ruleset is loaded.
- 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.

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 the `:network` keyword. The resulting CIDR network (e.g. 192.168.0.0/24) will be substituted into the rule when the ruleset is loaded.
- A set of addresses using a list.

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

```
nat on tl0 from 192.168.1.0/24 to any -> 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):

```
nat on tl0 from dc0/24 to any -> 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 then 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:

```
nat on tl0 from dc0/24 to any -> (tl0)
```

There is one major limitation to doing this: Only the first IP alias on an interface is evaluated when the interface name is placed in parentheses.

Bidirectional Mapping (1:1 mapping)

A bidirectional mapping can be established by using the `binat` rule. A `binat` 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` rules as they are with `nat` rules.

Example:

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

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

Translation Rule Exceptions

Exceptions can be made to translation rules by using the `no` keyword. For example, if the NAT example above was modified to look like this:

```
no nat on tl0 from 192.168.1.10 to any
nat on tl0 from 192.168.1.0/24 to any -> 24.2.74.79
```

Then the entire 192.168.1.0/24 network would have its packets translated to the external address 24.2.74.79 except for 192.168.1.10.

Note that the first matching rule wins; if it's a `no` rule, then the packet is not translated. The `no` keyword can also be used with `binat` and `rdr` rules.

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
TCP 192.168.1.35:2132 -> 24.5.0.5:53136 -> 65.42.33.245:22 TIME_WAIT:TIME_WAIT
UDP 192.168.1.35:2491 -> 24.5.0.5:60527 -> 24.2.68.33:53   MULTIPLE:SINGLE
```

Explanations (first line only):

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: Queueing] [Contents] [Next: Traffic Redirection]


www@openbsd.org

$OpenBSD: nat.html,v 1.10 2003/09/27 16:00:20 horacio Exp $

[Previous: Network Address Translation] [Contents] [Next: Packet Filtering]

PF: Redirection (rdr)

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:

```
rdr on tl0 proto tcp from any to any port 80 -> 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:

```
rdr on tl0 proto tcp from 27.146.49.0/24 to any port 80 -> \
   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:

```
rdr on tl0 proto tcp from 27.146.49.14 to any port 80 -> \
```

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

Redirection and Packet Filtering

Be aware that packets matching an `rdr` rule still must pass through the filtering engine and will be blocked or passed based on the filter rules that have been defined. Also be aware that packets matching an `rdr` rule will have their destination IP address and/or destination port changed to match the redirection address/port specified in the `rdr` rule. Consider this scenario:

- 192.0.2.1 - host on the Internet
- 24.65.1.13 - external address of OpenBSD router
- 192.168.1.5 - internal IP address of web server

Redirection rule:

```
rdr on tl0 proto tcp from 192.0.2.1 to 24.65.1.13 port 80 \
   -> 192.168.1.5 8000
```

Packet before the `rdr` rule is processed:

- Source address: 192.0.2.1
- Source port: 4028 (arbitrarily chosen by the operating system)
- Destination address: 24.65.1.13
- Destination port: 80

Packet after the `rdr` rule is processed:

- Source address: 192.0.2.1
- Source port: 4028
- Destination address: 192.168.1.5
- Destination port: 8000

The filtering engine will see the IP packet as it looks after `rdr` rules have been processed.

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

rdr on $ext_if proto tcp from any to $ext_if port 80 -> $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.

```
127.0.0.1:5000 stream tcp nowait nobody /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:

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

RDR and NAT Combination

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

```
rdr on $int_if proto tcp from $int_net to $ext_if port 80 -> \
   $server
no nat on $int_if proto tcp from $int_if to $int_net
nat on $int_if proto tcp from $int_net to $server port 80 -> \
   $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` rule above will cause the TCP/IP stack to see packets arriving on the internal interface with a destination address inside the internal network. To prevent the stack from issuing ICMP redirect messages (telling the client that its destination is reachable directly, breaking the reflection), disable redirects on the gateway, using

```
# sysctl -w net.inet.ip.redirect=0
# sysctl -w net.inet6.ip6.redirect=0 (if using IPv6)
```

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

[Previous: Network Address Translation] [Contents] [Next: Packet Filtering]


www@openbsd.org

$OpenBSD: rdr.html,v 1.9 2003/09/22 02:29:03 nick Exp $

[Previous: Traffic Redirection] [Contents] [Next: Logging]

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 is 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 end 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 int [af] [proto protocol] \
   from src_addr [port src_port] to dst_addr [port dst_port] \
   [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 specifies the `keep state` or `modulate state` option 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.

int
: The name of the network interface that the packet is moving through.

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. 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 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 the `:network` or `:broadcast` keywords. The resulting CIDR network (i.e., 192.168.0.0/24) or broadcast address (i.e., 192.168.0.255) will be substituted into the rule when the ruleset is loaded.
- 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
- 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.

`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 the SYN flag is "on".

`state`

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

- `keep state` - works with TCP, UDP, and ICMP.
- `modulate state` - works only with TCP. PF will generate strong Initial Sequence Numbers (ISNs) for packets matching this rule.

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
```

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 has the `keep state` option, 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. For example:

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

This allows any outbound TCP traffic on the `fxp0` interface and also permits the reply traffic to pass back through the firewall. While keeping state is a nice feature, its use 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.

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

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

Another advantage of keeping state is that corresponding ICMP traffic will be passed through the firewall. For example, if `keep state` is specified for a TCP connection 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.

It's important to note that stateful connections are limited to the interface they were created on. This is particularly important on routers and firewalls running PF, especially when a "default deny" policy is implemented as outlined above. If a firewall is keeping state on all outgoing connections on the external interface, those packets must still be explicitly passed through the internal interface.

Note that `nat`, `binat`, and `rdr` rules implicitly create state for matching connections as long as the connection is passed by the filter ruleset.

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 `/etc/pf.conf` file.

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
```

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.

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

The above rule 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 rule while a packet with SYN and ACK or just ACK would not.

Note: in previous versions of OpenBSD, the following syntax was supported:

```
. . . flags S
```

This is no longer true. A mask must now *always* be specified.

Flags are often used in conjunction with `keep state` rules to help control the creation of state entries:

```
pass out on fxp0 proto tcp all flags S/SA keep state
```

This would permit the creation of state on any outgoing TCP packet with the SYN flag set out of the SYN and ACK flags.

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:

```
. . . 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 FIN or SYN and RST). It's highly recommended to always `scrub` incoming traffic:

```
scrub in on fxp0
.
.
.
pass in on fxp0 proto tcp from any to any port ssh flags S/SA \
   keep state
```

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 from any to any 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 from any to any 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.

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. These addresses should be passed explicitly. Example:

```
pass in quick on lo0 all
```

```
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.

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"

# scrub incoming packets
scrub in all

# setup a default deny policy
block in  all
block out all

# pass traffic on the loopback interface in either direction
pass quick on lo0 all

# activate spoofing protection for the internal interface.
antispoof quick for $int_if inet

# 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 flags S/SA

# pass all traffic to and from the local network
pass in  on $int_if from $lan_net to any
pass out on $int_if from any to $lan_net

# pass tcp, udp, and icmp out on the external (Internet) interface.
# keep state on udp and icmp and modulate state on tcp.
pass out on $ext_if proto tcp all modulate state flags S/SA
pass out on $ext_if proto { udp, icmp } all keep state

# allow ssh connections in on the external interface as long as they're
# NOT destined for the firewall (ie, 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.
pass in log on $ext_if proto tcp from any to { !$ext_if, !$int_if } \
   port ssh flags S/SA keep state
```

[Previous: Traffic Redirection] [Contents] [Next: Logging]


www@openbsd.org

$OpenBSD: filter.html,v 1.13 2003/09/22 02:29:03 nick Exp $

[Previous: Packet Filtering] [Contents] [Next: Anchors and Named Rulesets]

PF: Logging

Table of Contents

Introduction

Packet logging in PF is done by pflogd(8) which listens on the pflog0 interface and writes packets to a log file (normally /var/log/pflog) in tcpdump(8) binary format. Filter rules that specify the `log` or `log-all` keyword are logged in this manner.

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, OpenBSD's 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`*.
- `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`, and `memory`.
- `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 two small shell scripts, 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.

First we have to create a user, `pflogger`, with a `/sbin/nologin` shell. The easiest way to create this user is with adduser(8).

After creating the user `pflogger`, create the following two scripts:

`/etc/pflogrotate`

```
FILE=/home/pflogger/pflog5min.$(date "+%Y%m%d%H%M")
kill -ALRM $(cat /var/run/pflogd.pid)
if [ $(ls -l /var/log/pflog | cut -d " " -f 8) -gt 24 ]; then
   mv /var/log/pflog $FILE
   chown pflogger $FILE
   kill -HUP $(cat /var/run/pflogd.pid)
fi
```

`/home/pflogger/pfl2sysl`

```
for logfile in /home/pflogger/pflog5min* ; do
   tcpdump -n -e -ttt -r $logfile | logger -t pf -p local0.info
   rm $logfile
done
```

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
```

Create a cron job for user `pflogger`:

```
# crontab -u pflogger -e
```

Add the following two lines:

```
# feed rotated pflog file(s) to syslog
0-59/5 * * * * /bin/sh /home/pflogger/pfl2sysl
```

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.

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 /var/run/pflogd.pid
/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 can take up to about 5-6 minutes (the cron job interval) before the logged packets appear in the file.

[Previous: Packet Filtering] [Contents] [Next: Anchors and Named Rulesets]


www@openbsd.org

$OpenBSD: logging.html,v 1.8 2003/09/22 02:29:03 nick Exp $

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

PF: Anchors and Named Rulesets

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 filter, `nat`, `rdr`, and `binat` rules.

Sub rulesets are attached to the main ruleset by using anchors. There are four types of `anchor` rules:

- `anchor` *name* - evaluates all filter rules in the anchor *name*
- `binat-anchor` *name* - evaluates all `binat` rules in the anchor *name*
- `nat-anchor` *name* - evaluates all `nat` rules in the anchor *name*
- `rdr-anchor` *name* - evaluates all `rdr` rules in the anchor *name*

Only the main ruleset can contain `anchor` rules.

Named Rulesets

A named ruleset is a group of filter and/or translation rules that has been assigned a name. An `anchor` point may contain more than one such ruleset. When PF comes across an `anchor` rule in the main ruleset, it will evaluate all the rulesets attached to that `anchor` point in alphabetical order according to their names. Processing will then continue in the main ruleset unless the packet matches a filter rule that uses the `quick` option or a translation rule within the anchor 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 all
```

```
pass  out on $ext_if all keep state
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*. To add rules to the *goodguys* anchor the following commands can be used:

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

This adds a `pass` rule to the ruleset named *ssh* attached to the *goodguys* anchor. PF will then evaluate this rule (and any other filter rules that get added) when it reaches the `anchor goodguys` line in the main ruleset.

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:www -f /etc/anchor-goodguys-www
```

This loads the rules from the `/etc/anchor-goodguys-www` file into the named ruleset *www* in the *goodguys* anchor.

Filter and translation rules can be loaded into a named ruleset using the same syntax and options as rules loaded into the main ruleset. One caveat, however, is that any macros that are used must also be defined within the named ruleset; macros that are defined in the main ruleset are *not* visible from named rulesets.

Each named ruleset, 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 named rulesets that are attached to that anchor. A named ruleset is not destroyed until it's flushed of all rules using pfctl(8). Once an anchor point has no named rulesets attached to it, it's also destroyed.

Anchor Options

Optionally, `anchor` rules can specify interface, protocol, source and destination address, etc., using the same syntax as filter 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 all
pass  out on $ext_if all keep state
anchor ssh in on $ext_if proto tcp from any to any 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:allowed -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.

Manipulating Named Rulesets

Manipulation of named rulesets is performed via `pfctl`. It can be used to add and remove rules from a ruleset without reloading the main ruleset.

To list all the rules in the ruleset *`allowed`* attached to the *`ssh`* anchor:

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

To flush all filter rules from the same ruleset:

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

If the ruleset name is omitted, the action applies to all rules in the anchor.

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

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


www@openbsd.org

$OpenBSD: anchors.html,v 1.7 2003/09/22 02:29:03 nick Exp $

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

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 `/etc/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 `/etc/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 } keep state"

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

PF Grammar

With the release of OpenBSD 3.3, PF's grammar has been tuned to allow for greater flexibility in a ruleset. PF is now able to infer certain keywords which means that they don't have to be explicitly stated in a rule. Ordering of keywords is also not as strict as it used to be.

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 all
```

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: Anchors and Sub Rulesets] [Contents] [Next: Address Pools and Load Balancing]


www@openbsd.org

$OpenBSD: shortcuts.html,v 1.7 2003/09/22 02:29:03 nick Exp $

OpenBSD

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

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 redirection address in `rdr` rules, as the translation address in `nat` rules, and as the target address in `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` rules, destination address for `rdr` 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.

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 list format.

In addition, the `static-port` keyword can be specified after one of the above methods to prevent `nat` rules from modifying the source port of TCP and UDP packets.

NAT Address Pool

An address pool can be used as the translation address in `nat` 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.

```
nat on $ext_if inet from any to any -> { 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.

```
nat on $ext_if inet from any to any -> 192.0.2.4/31 source-hash
```

This `nat` 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 }"

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

Successive connections will be redirected to the web servers in a round-robin manner.

As with the NAT example, if the web servers are all placed within a CIDR network block, the `source-hash` keyword can be used so that connections from a given IP address are always redirected to the same physical web server. Again, this is sometimes necessary to maintain session information while browsing a website.

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 $int_if route-to \
   { ($ext_if1 $ext_gw1), ($ext_if2 $ext_gw2) } round-robin \
   from $lan_net to any keep state
```

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. Return packets will be routed back to the same external interface that they exited (this is done by the ISPs) and will be routed back to the internal network normally.

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

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

```
nat on $ext_if1 from $lan_net to any -> ($ext_if1)
nat on $ext_if2 from $lan_net to any -> ($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
nat on $ext_if1 from $lan_net to any -> ($ext_if1)
nat on $ext_if2 from $lan_net to any -> ($ext_if2)

#  default deny
block in  from any to any
block out from any to any

#  pass all outgoing packets on internal interface
pass out on $int_if from any 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 tcp traffic from internal network.
pass in on $int_if route-to \
    { ($ext_if1 $ext_gw1), ($ext_if2 $ext_gw2) } round-robin \
    proto tcp from $lan_net to any flags S/SA modulate state
#  load balance outgoing udp and icmp traffic from internal network
pass in on $int_if route-to \
    { ($ext_if1 $ext_gw1), ($ext_if2 $ext_gw2) } round-robin \
    proto { udp, icmp } from $lan_net to any keep state

#  general "pass out" rules for external interfaces
pass out on $ext_if1 proto tcp from any to any flags S/SA modulate state
pass out on $ext_if1 proto { udp, icmp } from any to any keep state
pass out on $ext_if2 proto tcp from any to any flags S/SA modulate state
pass out on $ext_if2 proto { udp, icmp } from any to any keep state

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

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


www@openbsd.org

$OpenBSD: pools.html,v 1.6 2003/09/16 01:23:49 nick Exp $

[Previous: Address Pools and Load Balancing] [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 8Mbps, 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. Realtek 8139 (rl) based cards tend to be relatively poor performers while Intel 21143 (dc) 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 very large number of packets rapidly, at least if backed up with good NICs and a good ruleset.

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: Address Pools and Load Balancing] [Contents] [Next: Issues with FTP]


www@openbsd.org

$OpenBSD: perf.html,v 1.8 2003/09/22 02:29:03 nick 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. The FTP proxy used by OpenBSD and PF is ftp-proxy(8). To activate it, put something like this in the NAT section of `/etc/pf.conf`:

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

The explanation of this line is: "Traffic on the internal interface is redirected to the proxy server running on this machine which is listening at port 8021".

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/inetd.conf`:

```
127.0.0.1:8021 stream tcp nowait root /usr/libexec/ftp-proxy \
   ftp-proxy
```

and either rebooting the system or sending a 'HUP' signal to inetd(8). One way to send the 'HUP' signal is with the command:

```
kill -HUP `cat /var/run/inetd.pid`
```

You will note that ftp-proxy is listening on port 8021, the same port the above `rdr` statement is sending FTP traffic to. The choice of port 8021 is arbitrary, though 8021 is a good choice as it is not defined for any other application.

Please note that ftp-proxy(8) is to help **FTP clients** behind a PF filter; it is not used to handle an **FTP server** behind a PF filter.

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 from any to any port 21 keep state
pass in on $ext_if proto tcp from any to any port > 49151 \
   keep state
```

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. For the sake of discussion, we will assume the FTP server in question is again the standard OpenBSD ftpd(8), using the default range of ports.

Here is an example subset of rules which would accomplish this:

```
ftp_server = "10.0.3.21"

rdr on $ext_if proto tcp from any to any port 21 -> $ftp_server \
   port 21
```

```
rdr on $ext_if proto tcp from any to any port 49152:65535 -> \
   $ftp_server port 49152:65535

pass in quick on $ext_if proto tcp from any to $ftp_server \
   port 21 keep state
pass in quick on $ext_if proto tcp from any to $ftp_server \
   port > 49151 keep state
```

More Information on FTP

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

- FTP Reviewed

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


www@openbsd.org

$OpenBSD: ftp.html,v 1.8 2003/09/16 01:23:49 nick Exp $

[Previous: Issues with FTP] [Contents] [Next: Example #1: Firewall for Home or Small Office]

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 alters the pf(4) ruleset by adding rules to a named ruleset attached to an anchor point. Each time a user authenticates, authpf creates a new named ruleset and loads the preconfigured filter, `nat`, `binat`, and `rdr` rules into it. 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.

Linking Authpf into the Main Ruleset

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

```
nat-anchor authpf
rdr-anchor authpf
binat-anchor authpf
anchor authpf
```

Wherever the `anchor` rules are placed within the ruleset is where PF will branch off from the main ruleset to evaluate the authpf rules. It's not necessary for all four `anchor` rules to be present; for example, if authpf hasn't been setup to load any `nat` rules, the `nat-anchor` rule can be omitted.

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.

Filter and translation 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.

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`.

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`.

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
```

More Information

For a complete description of authpf operation, please refer to the authpf man page.

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"
dns_servers = "{ 10.0.1.56, 10.0.2.56 }"

pass in quick on $wifi_if proto udp from $user_ip to $dns_servers \
   port domain keep state
pass in quick on $wifi_if proto tcp from $user_ip to port { ssh, http, \
   https } flags S/SA keep state
```

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"
dns_servers = "{ 10.0.1.56, 10.0.2.56 }"

pass in quick on $wifi_if proto udp from $user_ip to $dns_servers \
   port domain keep state
pass in quick on $wifi_if proto tcp from $user_ip to $smtp_server \
   port smtp flags S/SA keep state
pass in quick on $wifi_if proto tcp from $user_ip to $pop3_server \
   port pop3 flags S/SA keep state
pass in quick on $wifi_if proto tcp from $user_ip to port { ssh, http, \
   https } flags S/SA keep state
```

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

```
# macros
wifi_if = "wi0"
ext_if  = "fxp0"

scrub in all

# filter
block drop all

pass out quick on $ext_if proto tcp from $wifi_if:network flags S/SA \
   modulate state
pass out quick on $ext_if proto { udp, icmp } from $wifi_if:network \
   keep state

pass in quick on $wifi_if proto tcp from $wifi_if:network to $wifi_if \
   port ssh flags S/SA keep state

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.
- Pass incoming SSH traffic from the wireless network destined for the gateway itself. This rule is necessary to permit users to log in.
- 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: Example #1: Firewall for Home or Small Office]


www@openbsd.org

$OpenBSD: authpf.html,v 1.3 2003/09/22 02:29:03 nick Exp $

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

PF: Example #1: Firewall for Home or Small Office

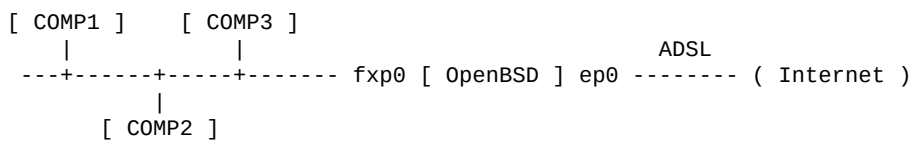
Table of Contents

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. This document will go through a complete ruleset that does just that.

The Network

The network is setup like this:



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. The internal network is using the 192.168.0.0 / 255.255.255.0 network block.

The OpenBSD router is a Pentium 100 with two network cards: a 3com 3c509B (ep0) and an Intel EtherExpress Pro/100 (fxp0). The router has an ADSL 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).
- 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 `net.inet.ip.forwarding` to "`1`".

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 = "fxp0"
ext_if = "ep0"

tcp_services = "{ 22, 113 }"
icmp_types = "echoreq"

priv_nets = "{ 127.0.0.0/8, 192.168.0.0/16, 172.16.0.0/12, 10.0.0.0/8 }"
```

The first two lines define the network interfaces that filtering will happen on. The third and fourth 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 permitted to reach the firewall machine. The last line defines the loopback and RFC 1918 address blocks.

Note: If the ADSL Internet connection required PPPoE, then filtering and NAT would have to take place on the `tun0` interface and *not* on `ep0`.

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 $ext_if
```

Scrub

There is no reason not to use the recommended scrubbing of all incoming traffic, so this is a simple one-liner:

```
scrub in all
```

Network Address Translation

To perform NAT for the entire internal network the following `nat` rule is used:

```
nat on $ext_if from $int_if:network to any -> ($ext_if)
```

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.

Redirection

The only redirection needed is for ftp-proxy(8) so that FTP clients on the local network can connect to FTP servers on the Internet.

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

Note that this rule will only catch FTP connections to port 21. If users regularly connect to FTP servers on other ports, then a list should be used to specify the destination port, for example: `from any to any port { 21, 2121 }`.

Filter Rules

Now the filter rules. Start with the default deny:

```
block all
```

At this point nothing will go through the firewall, not even from the internal network. The following rules will open up the firewall as per the objectives above as well as open up any necessary virtual interfaces.

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. In general, all traffic should be passed on the loopback interface. On OpenBSD, the loopback interface is lo(4).

```
pass quick on lo0 all
```

Next, the RFC 1918 addresses will be blocked from entering or exiting the external interface. These addresses should never appear on the public Internet, and filtering them will ensure that the router does not "leak" these addresses out from the internal network and also block any incoming packets with a source address in one of those networks.

```
block drop in  quick on $ext_if from $priv_nets to any
block drop out quick on $ext_if from any to $priv_nets
```

Note that `block drop` is used to tell PF not to respond with a TCP RST or ICMP Unreachable packet. Since the RFC 1918 addresses don't exist on the Internet, any packets sent to those addresses will never make it there anyways. The `quick` option is used to tell PF not to bother evaluating the rest of the filter rules if one of the above rules matches; packets to or from the `$priv_nets` networks will be immediately dropped.

Now open the ports used by those network services that will be available to the Internet:

```
pass in on $ext_if inet proto tcp from any to ($ext_if) \
   port $tcp_services flags S/SA keep state
```

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`.

ICMP traffic must now be passed:

```
pass in inet proto icmp all icmp-type $icmp_types keep state
```

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 some environments.

```
pass in on $int_if from $int_if:network to any keep state
```

The above rule will permit any internal machine to send packets through the firewall; however, it will *not* permit the firewall to initiate a connection to an internal machine. Is this a good idea? That depends on some of the finer details of the network setup. If the firewall is also a DHCP server, it may need to "ping" an address to verify its availability before assigning it. Permitting the firewall to connect to the internal network also allows someone who has ssh'ed into the firewall from the Internet to then access machines on the network. Keep in mind that *not* allowing the firewall to communicate directly to the network is not a large security benefit; if someone gets access to the firewall they can probably alter the filter rules anyways. By adding the following rule, the firewall will be able to initiate connections to the internal network:

```
pass out on $int_if from any to $int_if:network keep state
```

Note that if both of these lines are in place, the `keep state` option is not needed; all packets will be able to pass through the internal interface because there is a rule to pass packets in both directions. However, if the `pass out` line is *not* included, the `pass in` line *must* include `keep state`. There is also some performance benefit to keeping state: State tables are checked before rules are evaluated, and if a state match is found, the packet is passed through the firewall without going through ruleset evaluation. This can offer a performance benefit on a heavily loaded firewall, though in a system this simple it is unlikely to generate enough load to matter.

Finally, pass traffic out on the external interface:

```
pass out on $ext_if proto tcp all modulate state flags S/SA
pass out on $ext_if proto { udp, icmp } all keep state
```

TCP, UDP, and ICMP traffic is permitted to exit the firewall towards the Internet. State information is kept so that the returning packets will be passed in through the firewall.

The Complete Ruleset

```
# macros
int_if = "fxp0"
ext_if = "ep0"

tcp_services = "{ 22, 113 }"
icmp_types = "echoreq"

priv_nets = "{ 127.0.0.0/8, 192.168.0.0/16, 172.16.0.0/12, 10.0.0.0/8 }"

# options
set block-policy return
set loginterface $ext_if

# scrub
scrub in all

# nat/rdr
nat on $ext_if from $int_if:network to any -> ($ext_if)
rdr on $int_if proto tcp from any to any port 21 -> 127.0.0.1 \
   port 8021

# filter rules
block all

pass quick on lo0 all

block drop in  quick on $ext_if from $priv_nets to any
block drop out quick on $ext_if from any to $priv_nets

pass in on $ext_if inet proto tcp from any to ($ext_if) \
   port $tcp_services flags S/SA keep state

pass in inet proto icmp all icmp-type $icmp_types keep state

pass in  on $int_if from $int_if:network to any keep state
pass out on $int_if from any to $int_if:network keep state

pass out on $ext_if proto tcp all modulate state flags S/SA
pass out on $ext_if proto { udp, icmp } all keep state
```

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


www@openbsd.org

$OpenBSD: example1.html,v 1.9 2003/09/22 02:29:03 nick Exp $