Murus

OS X PF Manual

Index

This PF manual is directly derived from OpenBSD PF FAQ version 4.3. It is a very outdated PF version but it's the one more close to the current OS X Yosemite PF. Please note that Apple PF implementation is slightly different from OpenBSD 4.3 and some option may be unavailable.

The Murus Team

www.murusfirewall.com

PF: Getting Started

Table of Contents

Activation

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

pfctl -e
pfctl -d

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

Configuration

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

The pf.conf file has 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.
* 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 Load 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 -sd Show the current Dummynet rules (OS X 10.8• only)
pfctl -ss Show the current state table
pfctl -si Show filter stats and counters
pfctl -sa Show EVERYTHING it can show

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.

Lists can also contain nested lists:

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

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

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

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

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

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

Macros

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

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

```
ext_if = "fxp0"

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

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

Macros can also expand to lists, such as:

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

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

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

The macro $all_hosts now expands to 192.168.1.1, 192.168.1.2.

PF: Tables

Table of Contents

Introduction

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

* source and/or destination address in filter, scrub, NAT, and redirection rules.
* translation address in NAT rules.
* redirection address in redirection rules.
* destination address in route-to, reply-to, and dup-to filter rule options.

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

Configuration

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

* const - the contents of the table cannot be changed once the table is created. When this attribute is not specified, pfctl(8) may be used to add or remove addresses from the table at any time, even when running with a securelevel(7) of two or greater.
* persist - causes the kernel to keep the table in memory even when no rules refer to it. Without this attribute, the kernel will automatically remove the table when the last rule referencing it is flushed.

Example:

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

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

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

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

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

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

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

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

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

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

Manipulating with pfctl

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

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

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

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

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

 # pfctl -t spammers -T delete 218.70.0.0/16

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

Specifying Addresses

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

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

Address Matching

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

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

block in on dc0 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

PF: Packet Filtering

Table of Contents

Introduction

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

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

Rule Syntax

The general, highly simplified syntax for filter rules is:

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

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

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

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

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

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

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

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

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

protocol
The Layer 4 protocol of the packet:

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

src_addr, dst_addr

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

- A single IPv4 or IPv6 address.
- A CIDR network block.
- A fully qualified domain name that will be resolved via DNS when the ruleset is loaded. All resulting IP addresses will be substituted into the rule.
- The name of a network interface or group. Any IP addresses assigned to the interface will be substituted into the rule.
- The name of a network interface followed by /netmask (i.e., /24). Each IP address on the interface is combined with the netmask to form a CIDR network block which is substituted into the rule.
- The name of a network interface or group in parentheses ( ). This tells PF to update the rule if the IP address(es) on the named interface change. This is useful on an interface that gets its IP address via DHCP or dial-up as the ruleset doesn't have to be reloaded each time the address changes.
- The name of a network interface followed by any one of these modifiers:
  - :network - substitutes the CIDR network block (e.g., 192.168.0.0/24)
  - :broadcast - substitutes the network broadcast address (e.g., 192.168.0.255)
  - :peer - substitutes the peer's IP address on a point-to-point link

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

- A table.
- The keyword urpf-failed can be used for the source address to indicate that it should be run through the uRPF check.
- Any of the above but negated using the ! ("not") modifier.
- A set of addresses using a list.
- The keyword any meaning all addresses
- The keyword all which is short for from any to any.

src_port, dst_port

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

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

  - < (less than)

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

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

- o : (inclusive range)

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

tcp_flags

Specifies the flags that must be set in the TCP header when using proto tcp. Flags are specified as flags check/mask. For example: flags S/SA - this instructs PF to only look at the S and A (SYN and ACK) flags and to match if only the SYN flag is "on". In OpenBSD 4.1 and later, the default flags S/SA are applied to all TCP filter rules.

state

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

- keep state - works with TCP, UDP, and ICMP. In OpenBSD 4.1 and later, this option is the default for all filter rules.
- modulate state - works only with TCP. PF will generate strong Initial Sequence Numbers (ISNs) for packets matching this rule.
- synproxy state - proxies incoming TCP connections to help protect servers from spoofed TCP SYN floods. This option includes the functionality of keep state and modulate state.

Default Deny

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

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

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

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

Passing Traffic

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

Some examples:

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

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

The *quick* Keyword

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

Wrong:

```
block in on fxp0 proto tcp 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.

Keeping State

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

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

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

Starting in OpenBSD 4.1, all filter rules automatically create a state entry when a packet matches the rule. In earlier versions of OpenBSD the filter rule had to explicitly use the keep state option.

Example using OpenBSD 4.1 and later:

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

Example using OpenBSD 4.0 and earlier:

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

These rules allow 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. Starting with OpenBSD 3.5, the modulate state option can be used in rules that specify protocols other than TCP.

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

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

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

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

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

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

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 pf.conf file.

Stateful Tracking Options

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

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

no state
Prevents the rule from automatically creating a state entry.

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

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

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

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

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

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

An example rule:

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

The rule above defines the following behavior:

* Limit the absolute maximum number of states that this rule can create to

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

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

max-src-conn number
Limit the maximum number of simultaneous TCP connections which have completed the 3-way handshake that a single host can make.
max-src-conn-rate number / interval
Limit the rate of new connections to a certain amount per time interval.

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

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

overload <table>
Put an offending host's IP address into the named table.
flush [global]
Kill any other states that match this rule and that were created by this source IP. When global is specified, kill all states matching this source IP, regardless of which rule created the state.

An example:

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

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

TCP Flags

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

*** F : FIN - Finish; end of session**
*** S : SYN - Synchronize; indicates request to start session**
*** R : RST - Reset; drop a connection**
*** P : PUSH - Push; packet is sent immediately**
*** A : ACK - Acknowledgement**
*** U : URG - Urgent**
*** E : ECE - Explicit Congestion Notification Echo**
*** W : CWR - Congestion Window Reduced**

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

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

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

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

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.

In OpenBSD 4.1 and later, the default flags applied to TCP rules is flags S/SA. Combined with the OpenBSD 4.1 default of keep state on filter rules, these two rules become equivalent:

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

Each rule will match TCP packets with the SYN flag set and ACK flag clear and each will create a state entry for matching packets. The default flags can be overridden by using the flags option as outlined above.

In OpenBSD 4.0 and earlier there were no default flags applied to any filter rules. Each rule had to specify which flag(s) to match on and also had to explicity use the keep state option.

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

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

. . . flags S/FSRPAUEW bad idea!!

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

. . . flags S/SAFR

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

TCP SYN Proxy (unstable in OS X)

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

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

```
pass in on $ext_if proto tcp from any to $web_server port www \
  flags S/SA synproxy state
```

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

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

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

Blocking Spoofed Packets

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

PF offers some protection against address spoofing through the anti spoof 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 anti spoof rule would expand to:

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

These rules accomplish two things:

* Blocks all traffic coming from the 10.0.0.0/24 network that does not pass in through fxp0. Since the 10.0.0.0/24 network is on the fxp0 interface, packets with

a source address in that network block should never be seen coming in on any other interface.

* Blocks all incoming traffic from 10.0.0.1, the IP address on fxp0. The host machine should never send packets to itself through an external interface, so any incoming packets with a source address belonging to the machine can be considered malicious.

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

set skip on lo0

antispoof for fxp0 inet

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

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

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

Unicast Reverse Path Forwarding

Starting in OpenBSD 4.0, PF offers a Unicast Reverse Path Forwarding (uRPF) feature. When a packet is run through the uRPF check, the source IP address of the packet is looked up in the routing table. If the outbound interface found in the routing table entry is the same as the interface that the packet just came in on, then the uRPF check passes. If the interfaces don't match, then it's possible the packet has had its source address spoofed.

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

block in quick from urpf-failed label uRPF

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

uRPF provides the same functionality as antispoof rules.

Passive Operating System Fingerprinting

Passive OS Fingerprinting (OSFP) is a method for passively detecting the

operating system of a remote host based on certain characteristics within that host's TCP SYN packets. This information can then be used as criteria within filter rules.

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

```
# pfctl -s osfp
```

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

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

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

TAKE NOTE of the following:

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

IP Options

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

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

Filtering Ruleset Example

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

```
ext_if  = "fxp0"
int_if  = "dc0"
lan_net = "192.168.0.0/24"
# table containing all IP addresses assigned to the firewall
table <firewall> const { self }
# don't filter on the loopback interface
set skip on lo0
# scrub incoming packets
scrub in all
# setup a default deny policy
block all
# activate spoofing protection for all interfaces
block in quick from urpf-failed
# only allow ssh connections from the local network if it's from the
# trusted computer, 192.168.0.15. use "block return" so that a TCP RST is
# sent to close blocked connections right away. use "quick" so that this
# rule is not overridden by the "pass" rules below.
block return in quick on $int_if proto tcp from ! 192.168.0.15 \
   to $int_if port ssh
# pass all traffic to and from the local network.
# these rules will create state entries due to the default
# "keep state" option which will automatically be applied.
pass in  on $int_if from $lan_net to any
pass out on $int_if from any to $lan_net
# pass tcp, udp, and icmp out on the external (Internet) interface.
# tcp connections will be modulated, udp/icmp will be tracked
# statefully.
pass out on $ext_if proto { tcp udp icmp } all modulate state

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

```
# PF.
pass in log on $ext_if proto tcp from any to ! <firewall> \
   port ssh synproxy state
```

PF: Network Address Translation (NAT)

Table of Contents

Introduction

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

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

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

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

How NAT Works

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

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

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

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

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

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

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

NAT and Packet Filtering

NOTE: Translated packets must still pass through the filter engine and will be blocked or passed based on the filter rules that have been defined. The only exception to this rule is when the pass keyword is used within the nat rule. This will cause the NATed packets to pass right through the filtering engine.

Also be aware that since translation occurs before filtering, the filter engine will see the translated packet with the translated IP address and port as outlined in How NAT Works.

IP Forwarding

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

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

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

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

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

Configuring NAT

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

```
nat [pass] [log] on interface [af] from src_addr [port src_port] to \
  dst_addr [port dst_port] -> ext_addr [pool_type] [static-port]
```

nat
The keyword that begins a NAT rule.

pass
Causes translated packets to completely bypass the filter rules.

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

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

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

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

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

ruleset is loaded. All resulting IP addresses will be substituted into the rule.

- The name or group of a network interface. Any IP addresses assigned to the interface will be substituted into the rule at load time.
- The name of a network interface followed by /netmask (e.g. /24). Each IP address on the interface is combined with the netmask to form a CIDR network block which is substituted into the rule.
- The name or group of a network interface followed by any one of these modifiers:
  - o :network - substitutes the CIDR network block (e.g., 192.168.0.0/24)
  - o :broadcast - substitutes the network broadcast address (e.g., 192.168.0.255)
  - o :peer - substitutes the peer's IP address on a point-to-point link

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

- A table.
- Any of the above but negated using the ! ("not") modifier.
- A set of addresses using a list.
- The keyword any meaning all addresses

src_port

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

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

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

  - o : (inclusive range)

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

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

dst_addr

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

dst_port

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

ext_addr

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

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

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

- A set of addresses using a list.

pool_type

Specifies the type of address pool to use for translation.

static-port

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

This would lead to a most basic form of 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:network 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, 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:network to any -> (tl0)

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

Bidirectional Mapping (1:1 mapping)

A bidirectional mapping can be established by using the binat 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 192.168.1.208 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 192.168.1.208.

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

Explanations (first line only):

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

TCP
The protocol being used by the connection.

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

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

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

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

PF: Redirection (Port Forwarding)

Table of Contents

Introduction

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

Let's look at an example:

```
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
```

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

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

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

Redirection and Packet Filtering

NOTE: Translated packets must still pass through the filter engine and will be blocked or passed based on the filter rules that have been defined.

The only exception to this rule is when the pass keyword is used within the rdr rule. In this case, the redirected packets will pass statefully right through the filtering engine: the filter rules won't be evaluated against these packets. This is a handy shortcut to avoid adding pass filter rules for each redirection rule. Think of it as a normal rdr rule (with no pass keyword) associated to a pass filter rule with the keep state keyword. However, if you want to enable more specific filtering options such as synproxy, modulate state, etc. you'll still have to use a dedicate pass rule as these options don't fit into redirection rules.

Also be aware that since translation occurs before filtering, the filter engine will see the translated packet as it looks after it's had its 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 port 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 filter engine will see the IP packet as it looks after translation has taken place.

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

PF: Shortcuts For Creating Rulesets

Table of Contents

Introduction

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

Using Macros

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

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

port numbers, IP addresses, and interface names, is recommended practice.

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

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

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

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

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

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

Using Lists

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

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

Now look at the following simplification:

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

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

```
NoRouteIPs = "{ 127.0.0.0/8, 192.168.0.0/16, 172.16.0.0/12, \
  10.0.0.0/8 }"
ExtIF = "tl0"
```

```
block in  quick on $ExtIF from $NoRouteIPs to any
block out quick on $ExtIF from any to $NoRouteIPs
```

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

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

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

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

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

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

Elimination of Keywords

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

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

This can now be reduced to:

```
block 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.

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

NOTE: In OpenBSD 3.7 and later, the behavior of runtime options has changed. Previously, once an option was set it was never reset to its default value, even if the ruleset was reloaded. Starting in OpenBSD 3.7, whenever a ruleset is loaded, the runtime options are reset to default values before the ruleset is parsed. Thus, if an option is set and is then removed from the ruleset and the ruleset reloaded, the option will be reset to its default value.

set block-policy option

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

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

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

set debug option

Set pf's debugging level.

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

The default is urgent.

set fingerprints file

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

set limit option value

Set various limits on pf's operation.

- frags - maximum number of entries in the memory pool used for packet reassembly (scrub rules). Default is 5000.
- src-nodes - maximum number of entries in the memory pool used for tracking source IP addresses (generated by the sticky-address and source-track options). Default is 10000.
- states - maximum number of entries in the memory pool used for state

table entries (filter rules that specify keep state). Default is 10000.
- tables - maximum number of tables that can be created. The default is 1000.
- table-entries - the overall limit on how many addresses can be stored in all tables. The default is 200000. If the system has less than 100MB of physical memory, the default is set to 100000.

set loginterface interface

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

set optimization option

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

The default is normal.

set ruleset-optimization option

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

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

set skip on interface

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

set state-policy option

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

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

The default is floating.

set timeout option value

Set various timeouts (in seconds).

- interval - seconds between purges of expired states and packet fragments. The default is 10.
- frag - seconds before an unassembled fragment is expired. The default is 30.
- src.track - seconds to keep a source tracking entry in memory after the last state expires. The default is 0 (zero).

Example:

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

PF: Scrub (Packet Normalization)

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 a 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. The no keyword can be used in front of scrub to specify packets that will not be scrubbed. Just as with nat rules, the first matching rule wins.

More on the principle and concepts of scrubbing can be found in the Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-End Protocol Semantics paper.

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 packets with random values to compensate for operating systems that use predictable values. This option only applies to 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.

reassemble tcp
Statefully normalizes TCP connections. When using scrub reassemble tcp, a direction (in/out) may not be specified. The following normalizations are performed:

- Neither side of the connection is allowed to reduce their IP TTL. This is done to protect against an attacker sending a packet such that it reaches the firewall, affects the held state information for the connection, and expires before reaching the destination host. The TTL of all packets is raised to the highest value seen for the connection.
- Modulate RFC1323 timestamps in TCP packet headers with a random number. This can prevent an observer from deducing the uptime of the

host or from guessing how many hosts are behind a NAT gateway.

Examples:

```
scrub in on fxp0 all fragment reassemble min-ttl 15 max-mss 1400
scrub in on fxp0 all no-df
scrub    on fxp0 all reassemble tcp
```

PF: Anchors

Table of Contents

Introduction

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

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

Anchors

An anchor is a collection of filter and/or translation rules, tables, and other anchors that has been assigned a name. When PF comes across an anchor rule in the main ruleset, it will evaluate the rules contained within the anchor point as it evaluates rules in the main ruleset. Processing will then continue in the main ruleset unless the packet matches a filter rule that uses the quick option 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. Anchors can be populated with rules by three methods:

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

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

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

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

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

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

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

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

To load rules directly from the main ruleset, enclose the anchor rules in a

brace-delimited block:

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

Inline anchors can also contain more anchors.

```
allow = "{ 192.0.2.3 192.0.2.4 }"

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

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

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

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

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

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

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

Anchor Options

Optionally, anchor rules can specify interface, protocol, source and destination address, tag, etc., using the same syntax as 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 -f -
```

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

The same syntax can be applied to inline anchors.

```
allow = "{ 192.0.2.3 192.0.2.4 }"

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

Manipulating Anchors

Manipulation of anchors is performed via pfctl. It can be used to add and

remove rules from an anchor without reloading the main ruleset.

To list all the rules in the anchor named ssh:

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

To flush all filter rules from the same anchor:

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

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

PF: Logging

Table of Contents

Introduction

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

Logging Packets

In order to log packets passing through PF, the log keyword must be used within NAT/rdr and filter rules. Note that PF can only log packets that it's blocking or passing; you cannot specify a rule that only logs packets.

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

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

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

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

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

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

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

Reading a Log File

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

To view the log file:

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

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

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

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

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

Filtering Log Output

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

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

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

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

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

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

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

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

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

Example:

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

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

PF: Issues with FTP

Table of Contents

FTP Modes

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

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

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

With passive mode FTP (the default mode with OpenBSD's ftp(1) client), the client requests that the server pick a random port to listen on for the data connection. The server informs the client of the port it has chosen, and the client connects to this port to transfer the data. Unfortunately, this is not always possible or desirable because of the possibility of a firewall in front

of the FTP server blocking the incoming data connection. OpenBSD's ftp(1) uses passive mode by default; to force active mode FTP, use the -A flag to ftp, or set passive mode to "off" by issuing the command "passive off" at the “ftp>" prompt.

FTP Client Behind the Firewall

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

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

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

```
nat-anchor "ftp-proxy/*"
rdr-anchor "ftp-proxy/*"
rdr on $int_if proto tcp from any to any port 21 -> 127.0.0.1 \
  port 8021
```

The first two lines are a couple anchors which are used by ftp-proxy to add rules on-the-fly as needed to manage your FTP traffic. The last line redirects FTP from your clients to the ftp-proxy(8) program, which is listening on your machine to port 8021.

You also need an anchor in the rules section:

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

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

```
ftpproxy_flags=""
```

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

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

To enable active mode connections, you need the '-r' switch on ftp-proxy(8) (for this you had to run the old proxy with "-u root").

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. In order to accomplish this, we turn again to ftp-proxy(8).

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

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

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

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

Now for the pf.conf rules:

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

nat-anchor "ftp-proxy/*"
nat on $ext_if inet from $int_if -> ($ext_if)
rdr-anchor "ftp-proxy/*"

pass in on $ext_if inet proto tcp to $ext_ip port 21 \
   flags S/SA keep state
```

```
pass out on $int_if inet proto tcp to $ftp_ip port 21 \
   user proxy flags S/SA keep state
anchor "ftp-proxy/*"
```

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

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

Proxying TFTP

Trivial File Transfer Protocol (TFTP) suffers from some of the same limitations as FTP does when it comes to passing through a firewall. Luckily, PF has a helper proxy for TFTP called tftp-proxy(8).
tftp-proxy(8) is setup in much the same way as ftp-proxy(8) was in the FTP Client Behind the Firewall section above.

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

anchor "tftp-proxy/*"
```

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

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

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

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