The OpenBSD Packet Filter HOWTO

Wouter Coene

version 20020405.2, generated April 5, 2002
available at `http://www.inebriated.demon.nl/pf-howto/`

Contents

1 Introduction

The OpenBSD Packet Filter (OpenBSD `PF`) is the stateful firewall package that is part of the OpenBSD kernel since OpenBSD 3.0. This document describes how to set up and manage PF rulesets and NAT mappings.

1.1 Intended audience

The intended audience for this document are system- and network administrators with at least a basic knowledge of (inter)networking and the network protocols involved. Knowledge of other firewall systems is not required, but can help in mastering the more complex topics.

1.2 Status

This document is currently a work-in-progress, so not everything regarding OpenBSD PF may be covered yet.

Todo list:

- Network Address Translation
- IPv6
- more notes regarding the extensive feature set of the pfctl command
- AuthPF

1.3 Copyright and license

The OpenBSD Packet Filter HOWTO version 20020405.2
Copyright (C) Wouter Coene, 2001, 2002.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

- The name of the author may not be used to endorse or promote products derived from this software without specific prior written permission.
- This software may not be mass-distributed for sale without informing the author at least two weeks in advance.

THIS DOCUMENT IS PROVIDED BY THE AUTHOR "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

1.4 Contacting the author

You can contact the Wouter Coene through email at wouter@inebriated.demon.nl, or via snail-mail:

```
De Deel 17
3471 EN Kamerik
The Netherlands
```

2 Basic firewalling

A firewall is supposed to protect a network from potential attacks originating from another network. It does this by inspecting packets traveling between the two networks, and imposing restraints upon the types, targets and even content of these packets.

This section explains how to set up firewalling using PF. For this section I'll use a corporate network and firewall as an example. The network itself will consist of a `TCP/IP` network with network address `260.250.1.0/24`. There will be a corporate webserver running on `260.250.1.3`, and a firewall with two network interfaces, `xl0` and `xl1`. The corporate network is connected to the `xl1` interface, and the internet uplink is connected to `xl0`.

2.1 Ruleset basics

The firewalling component of PF uses a set of rules which describe actions to be taken for certain packets. These rules are read from a file and loaded into the OpenBSD kernel using the `pfctl` command (for more information on loading your ruleset, see section 2.11).

Such a rule file might look like the following:

```
block in all
pass  in all
```

Let's analyze what happens here. The first rule tells PF to block all incoming packets. Unlike some other firewalls, PF doesn't stop rule parsing when it finds a match. Instead, it notes the fact that it is planning on blocking the packet, and moves on to the next rule.

The next rule tells PF to pass all incoming packets to whatever is their destination. Again, PF notes the fact that it is planning on passing the packet, and moves on to the next rule.

Since there is no next rule, PF starts looking at what it was planning to do. In this case, the last rule that matched told PF to pass the packet, so it does.

Well, that doesn't look very useful, so let's try something more interesting. Let's allow access to the corporate webserver running at IP address `260.250.1.3`[1]. You might try something like this:

[1] A completely fictional example

```
block in all
pass  in from any to 260.250.1.3/32
```

Again, the first rule tells PF to note that it should plan on blocking this packet unless it matches any other rules.

The second rule tells PF to pass packets that have their destination set to `260.250.1.3`, where the '/32' denotes to PF that it should match the address on the full 32 bits.

But what about return traffic, you might ask. Well, that's fairly simple, simply allow traffic from `260.250.1.3` to go anywhere:

```
block in all
pass  in from any to 260.250.1.3/32
pass  in from 260.250.1.3/32 to any
```

2.2 More advanced rulesets

Suppose it is decided that another website will be run on the corporate webserver, this one containing sensitive corporate information, so it shouldn't be accessible from the outside. This website will, contrary to the other one, run on the non-standard `TCP` port `8000`.

So now you'll not only have to filter on destination address, but on the `TCP` port number as well, to make sure nobody can connect to port `8000` from the outside and access the sensitive data:

```
block in all
pass  in proto tcp from any to 260.250.1.3/32 port = 80
pass  in proto tcp from 260.250.1.3/32 port = 80 to any
```

As you can see, we're not only filtering on the port number, we're also telling PF to only allow packets of protocol `TCP` to pass through to the firewall to the corporate webserver. This is necessary, since the `IP` protocol itself doesn't know about port numbers. Only `TCP` and `UDP` can differentiate between different ports.

2.3 Keeping state

The PF firewall component is capable of remembering what `TCP`, `UDP` or `ICMP` sessions were created, and can filter packets according to this session table. This is called *keeping state*.

Whenever PF sees a packet match a rule that instructs PF to keep state, it creates a new entry in the state table based upon the information in the packet. This results in subsequent packets from the same session being passed along without going through the rule matching phase.

Keeping state on TCP connections involves the careful checking of TCP sequence numbers of packets against the state table, and dropping packets that don't match the state of the connection, thus decreasing the possibility that hosts behind the firewall lacking a good TCP stack implementation are taken advantage of. When keeping state on UDP sessions, PF allows a single return packet for each packet matching the rule which creates the state table entry.

Suppose it is decided that web-browsing should be possible from our corporate network, which requires passing both TCP connections to port 80 as well as allowing DNS requests on port 53.

Using the PF state engine for this is a safe way of allowing this without at the same time opening the entire network to outside attacks. While you're at it, you might as well use the state engine for the access to the corporate webserver as well, resulting in more secure access control to that server:

```
block in all
pass  in proto udp from 260.250.1.0/24 to any port = 53 keep state
pass  in proto tcp from 260.250.1.0/24 to any port = 80 keep state
pass  in proto tcp from any to 260.250.1.3/32 port = 80 keep state
```

The third rule enables hosts on the corporate network to make connections to the HTTP port on external hosts, instructing PF to create an entry in its state table for these connections. The last rule instructs PF to do the same for connections made from outside hosts to the corporate webserver, thus making obsolete the rule we needed for return traffic when we weren't using the state engine.

Using the state engine might seem like burdening the firewall with extra load, only slowing down traffic. However, state table lookups under PF are much faster than ruleset parsing. A typical ruleset of 50 rules takes about 50 rule comparisons, whereas a state table of 50,000 entries, due to its binary tree structure, takes only about 16 comparisons. This fact, combined with the added security, makes it more than worth using the state engine even for the more simple tasks, which could have easily been done without it.

2.4 Breaking from rulesets: the `quick` keyword

Sometimes it might be appropriate to have PF immediately stop parsing the ruleset and do whatever it should do whenever a packet matches a specific rule. For this, PF has the `quick` keyword. A matching rule that has the `quick` option set will result in the termination of ruleset parsing for the matching packet.

This is especially useful for protecting your network against spoofed packets, as the following example shows:

```
block in all
block in quick from 10.0.0.0/8 to any
block in quick from 172.16.0.0/12 to any
block in quick from 192.168.0.0/16 to any
block in quick from 255.255.255.255/32 to any
pass  in all
```

This ruleset tells PF to immediately drop packets originating from `10.0.0.0/8`, `172.16.0.0/12`, `192.168.0.0/16` and `255.255.255.255/32`. Other packets are passed through.

This also results in a big performance boost if used appropriately for rules catching lots of traffic, as PF won't try to match any subsequent rules.

2.5 Matching network interfaces

It is also possible to match the network interface on which a packet is received. Let's adapt our previous anti-spoofing example to this, keeping our corporate network structure in mind:

```
block in all
block in quick on xl0 from 10.0.0.0/8 to any
block in quick on xl0 from 172.16.0.0/12 to any
block in quick on xl0 from 192.168.0.0/16 to any
block in quick on xl0 from 255.255.255.255/32 to any
pass  in all
```

2.6 Matching `TCP` flags

To enable the blocking of invalid packets, you can also instruct PF to filter on `TCP` flags using the `flags` keyword, followed by a list of flags to match on, an optional forward slash, and an optional flag mask.

PF will, for every `TCP` packet, first clear everything but the flags that were specified in the mask, and then match on the flags that should be matched on. So saying '`flags S/SA`' instructs PF to first mask out everything but the `SYN` and `ACK` flags, and then check if SYN is set.

The following flags are recognized:

F : `FIN`, for closing connections

S : `SYN`, for opening connections

R : `RST`, for connection resets

P : `PSH`, for making sure all data has arrived

A : `ACK`, for acknowledgement packets

U : `URG`, indicating this packet is urgent

For example, a packet requesting a new connection sets only the `SYN` flag, and a packet acknowledging a connection sets both `SYN` and `ACK`, whereas a packet indicating a refused connection sets both `ACK` and `RST`.

Using an invalid combination of `TCP` flags is a popular way to secretly scan hosts for open ports. Using the `flags` keyword, you can defend your system against these secretive scans, and force port scanners to use scanning methods that are more easily detectable.

Let's take our state-keeping example from earlier in this HOWTO. We want to enforce that only `TCP` packets, which of the `SYN` and `ACK` flags only have `SYN` set, get an entry in the state table:

```
block in all
pass  in proto udp from 260.250.1.0/24 to any port = 53 keep state
pass  in proto tcp from 260.250.1.0/24 to any port = 80 \
                                    flags S/SA keep state
pass  in proto tcp from any to 260.250.1.3/32 port = 80 \
                                    flags S/SA keep state
```

This should prevent the port scanning techniques mentioned above from passing our firewall.

2.7 Sets

It is possible to, instead of specifying a single source or destination host, specify a set of hosts. This is done by enclosing the hosts in curly braces, and by separating them by commas.

So if your old ruleset had rules in it like this:

```
block in quick on xl0 from 10.0.0.0/8 to any
block in quick on xl0 from 172.16.0.0/12 to any
block in quick on xl0 from 192.168.0.0/16 to any
block in quick on xl0 from 255.255.255.255/32 to any
```

You can replace them with a single rule:

```
block in quick on xl0 from { 10.0.0.0/8, 172.16.0.0/12, \
              192.168.0.0/16, 255.255.255.255/32 } to any
```

This can also be done for interfaces, protocols, and ports. The `pfctl` program will split up such rules into one rule for each combination, so PF can optimize your ruleset with the technique described in section 2.9. Additionally, it increases readability by orders of magnitude for large sets of hosts, interfaces, protocols or ports.

2.8 Variable expansion

PF also supports variable expansion, modelled after that of the shell. Variables are defined by assigning them a value, and expanded by prepending the variable name with a dollar sign ('$'):

```
webserver="260.250.1.3/32"
pass  in from any to $webserver port = 80 keep state
```

The value you want to assign to the variable must be quoted.

2.9 Ruleset optimization: skip steps

Unlike IPFilter, OpenBSD PF doesn't support the `group` keyword. The OpenBSD PF developers have chosen a scheme called *skip steps*, in which rulesets are optimized automatically.

Consider a ruleset looking like this:

```
block in quick on xl0 from 10.0.0.0/8 to any
block in quick on xl0 from 172.16.0.0/12 to any
block in quick on xl0 from 192.168.0.0/16 to any
block in quick on xl0 from 255.255.255.255/32 to any
```

For each incoming packet, this ruleset is evaluated from top to bottom. Imagine a packet is received on interface `xl1`. The first rule is evaluated, but is found not to match. Now, since the other rules also apply to interface `xl0`, PF can safely skip these rules.

When you load a ruleset the following parameters are compared between successive rules (in this order):

1. interface
2. protocol
3. source address
4. source port
5. destination address
6. destination port

For each rule, PF automatically calculates a so-called *skip step* for each of these parameters, which tells PF how many successive rules have the same value for the parameter.

If an incoming packet on interface `xl1` is matched against our example ruleset, PF notices that the packet's incoming interface didn't match the one in the first rule, and since the next 3 rules also mention that interface, it skips these rules altogether.

So if you'd like to maximize your ruleset performance, you should sort your ruleset by interface, by protocol, source address and port, and finally by destination address and port, in that order.

2.10 Putting it all together

Let's put together all that we have learned about PF using our earlier example of the corporate network. The following ruleset is the result:

```
# set up some variables
external="xl0"
internal="xl1"
corporate="260.250.1.0/24"
webserver="260.250.1.3/32"
private="{ 10.0.0.0/8, 172.16.0.0/12, 192.168.0.0/16, \
                                  255.255.255.255/32 }"
```

```
# block by default
block in all

# allow web-browsing from the corporate network
pass  in quick on $internal proto udp from $corporate to any \
                                           port = 53 keep state
pass  in quick on $internal proto tcp from $corporate to any \
                               port = 80 flags S/SA keep state

# drop spoofed packets
block in quick on $external from $private to any

# allow access to the corporate webserver from the internet
pass  in quick on $external proto tcp from any to $webserver \
                               port = 80 flags S/SA keep state
```

There you have it, a secure firewall. Of course, this is not the only thing neccesary for securing something like a corporate network. But for a first line of defense, it's doing its job quite well.

2.11 Loading your ruleset

Once you're happy with your ruleset, save it as `/etc/pf.conf`, then either reboot your machine, or execute

```
pfctl -R /etc/pf.conf
```

To have your rule set loaded automatically during the OpenBSD boot sequence, remember to set `pf=YES` in `/etc/rc.conf`.

3 Filtering Bridges

A bridge, or repeater, is a network device that connects two or more network segments together. It is commonly a simple box, which just repeats any incoming packets on the other network segment(s). However, an OpenBSD server can also be used for bridging, which makes it possible to use PF to filter traffic between the network segments.

This section explains how to set up a filtering bridge using OpenBSD PF. The network used in the examples is a university network, `IPv4` network address `10.0.0.0/8`, consisting of a large number of student workstations, a webserver on `10.0.0.1/32`, and a shell server on `10.0.0.2/32`.

For security reasons, the university staff would like to separate the network segment the students connect to from the server network segment, and filter the traffic in between, with minimal changes to the network topology.

The decision is made to use an OpenBSD server, with the student network segment connected to interface `xl0`, and the server network segment connected to interface `xl1`.

3.1 Two directions

Packets traversing the bridge go through PF twice: they go in through one interface, and come out through the other. So our ruleset must allow both incoming and outgoing traffic, causing us to start with the following ruleset:

```
pass in  on xl0 any
pass out on xl0 any
pass in  on xl1 any
pass out on xl1 any
```

This will allow traffic received on both `xl0` and `xl1` to enter the bridge, and allow said traffic to be sent to the correct network segment.

3.2 Stateful filtering

The OpenBSD Packet Filter has this really nice feature called *keeping state*, described in section 2.3, which can be used in the example network to increase the security of the server network segment.

However, there's one thing we have to keep in mind when using stateful filtering: the entries in the state table are indexed by a key consisting of the source- and destination addresses and `TCP` ports, where the order of these two pairs is relevant. If an outgoing packet from `A` to `B` creates an entry in this state table, PF will pass outgoing packets from `A` to `B`, and incoming packets from `B` to `A`. It will still block outgoing packets from `B` to `A`, and incoming packets from `A` to `B`, which in the non-bridged case is perfectly clear and obvious.

However, when using state keeping on a bridge, the packet goes through PF twice; it is an incoming packet on one interface, and an outgoing packet on the other.

There are two solutions to this problem. One is to create two entries in the state table for each connection, using two rules with the `keep state` option. This, however, increases the load on the bridge server, and is not recommended, since the other option is fairly simple and elegant:

From the perspective of PF, packets go through the bridge twice. If you're looking at one interface, you'll see exactly the same traffic, only the direction is reversed. Therefore, we can ignore one interface and do all the filtering on the other.

We'd like to keep state on connections to both the webserver and the shell server, and since we trust the student network the least[2], we'll filter on the `xl0` interface, simply passing along all traffic on the `xl1` interface:

```
# some variables
web="10.0.0.1/32"
shell="10.0.0.2/32"

# allow all traffic traversing xl1
pass  in  quick on xl1 all
pass  out quick on xl1 all

# block traffic on xl0 by default
block in  on xl0 all
block out on xl0 all

# allow connections to the web- and shell server
pass  in  quick on xl0 proto tcp from any to $web \
                    port = 80 flags S/SA keep state
pass  in  quick on xl0 proto tcp from any to $shell \
              port = { 22, 23 } flags S/SA keep state
```

[2]A purely psychological reason

4 Firewalling tricks

To increase the security of the host(s) it is supposed to protect, OpenBSD PF has a number of unique features to correct mistakes in `TCP/IP` stack implementations, which are described in this section.

4.1 State modulation

To ensure proper delivery of `TCP` packets and to prevent connection hijacking, the `TCP` protocol utilizes a sequence numer scheme in which a random initial sequence number (ISN) is chosen at the start of a connection, which is incremented for each byte transmitted. However, many popular `TCP` implementations use a very poor random number generator for generating these ISNs[3], thus making it more likely `TCP` connections originating from such systems could be taken over by malicious people.

That is why the OpenBSD PF developers chose to add *state modulation*. This involves generating a more random initial sequence number for connections matching a PF rule, and translating the sequence numbers of packets passing the firewall from the ISN generated by the host to the ISN generated by the firewall and vice-versa.

This can be done by adding the `modulate state` keyword to PF rules, such as this one, protecting the corporate network defined in the previous chapter:

```
pass  in quick on xl1 proto tcp from 260.250.1.0/24 to any \
                                   flags S/SA modulate state
```

The `modulate state` option implies `keep state`, described in section 2.3.

4.2 Packet normalization

Since some `IP` stacks don't correctly implement `IP` packet defragmentation, OpenBSD PF provides the `scrub` directive. If a `scrub` rule matches a packet, the PF normalization component makes sure the packet is defragmented and completely stripped of all abnormalities before it is sent along to its final destination[4].

[3]For more information about ISN generation, along with a survey of the ISN generation on some popular operating systems, see `http://razor.bindview.com/publish/papers/tcpseq.html`

[4]At the time of writing, it is not entirely clear to me how this interacts with state keeping. Could any of the PF developers comment on this?

Normalizing all incoming network traffic would require a rule such as this:

```
scrub in all
```

Using the `scrub` directive uses quite an amount of server resources, so its use should be limited to protecting only the weak `TCP/IP` stack implementations.

Additional options that apply to the `scrub` directive are:

no-df clear the `don't fragment` bit from a matching `IP` packet.

min-ttl *number* enforce a minimum *time to live* for matching `IP` packets, dropping packets that don't match the requirement.

5 Migrating from IPFilter

The ruleset model OpenBSD PF uses was modelled after that of IPFilter. There are also quite a few differences, which this section tries to document.

5.1 `head` and `group` are gone

The **`head`** and **`group`** keywords, which were used in IPFilter to group a number of rules, are no longer needed under OpenBSD PF. If you used to use **`head`** and **`group`**, you'll have to manually re-order your rulesets so they'll work under OpenBSD PF.

OpenBSD PF has an automatic scheme for ruleset optimization, called *skip step*. See section 2.9 for more information.

6 Other documentation

There are a number of sources for information on OpenBSD PF and firewalls in general available on the Internet. Here's a short summary of stuff that might be interesting:

`http://www.benzedrine.cx/pf.html` The original homepage of what is now OpenBSD PF.

`http://www.obfuscation.org/ipf/` The IPFilter HOWTO. Though the HOWTO you're reading right now tries to be as complete as possible with regard to OpenBSD PF, it might be interesting to look at OpenBSD PF's roots too.

`http://www.linuxdoc.org/HOWTO/Firewall-HOWTO.html` The Linux Firewall and Proxy HOWTO, which also covers the subject of setting up user-space proxies such as Squid and SOCKS. Written for Linux, but might be of interest to OpenBSD users as well.

`http://www.openbsd.org/cgi-bin/man.cgi?query=pfctl&sektion=8&format=html` OpenBSD manual page on the `pfctl` program.

`http://www.openbsd.org/cgi-bin/man.cgi?query=pf.conf&sektion=5&format=html` OpenBSD manual page on the format of OpenBSD PF rulesets.

`http://www.openbsd.org/cgi-bin/man.cgi?query=pf&sektion=4&format=html` OpenBSD manual page on the pf device. Mainly of interest for developers.

7 Thanks

The author would like to thank the following people for providing help with some of the more complicated subjects, for clarifying some of the internal workings of OpenBSD PF, for pointing out errors or mistakes in previous versions of this document, or generally for making suggestions (in alphabetical order):

- Matthew Clarke <Matthew_Clarke@mindlink.bc.ca>
- Mike Frantzen <frantzen@w4g.org>
- Markus Friedl <markus@openbsd.org>
- Artur Grabowski <art@blahonga.org>
- Daniel Hartmeier <daniel@benzedrine.cx>
- Erik Liden <erik@ipunplugged.com>
- Rod Whitworth <listener@witworx.com>
- Jim Zajkowski <jim@jimz.net>