Handling FreeBSD's latest firewall semantics and frameworks

by Adrian PENIŞOARĂ*

Abstract

Despite having a powerful firewall service in its base system since early versions, known as the IPFW facility, FreeBSD has imported over time another popular packet filtering framework, the IPF system, and now has just imported the new kid on the packet filtering block, namely the open PF framework raised on the OpenBSD grounds.

Having three packet filtering frameworks at your feet might sound delightful, but actually choosing one of them or making them cooperate might give you the shivers. Each framework has its strengths and weaknesses and often has it's own different idea on how to handle certain aspects.

Note: an extended version of this article will appear on the conference's website.

1 Introduction

1.1 The players

For quite some time the FreeBSD users have been stuck with the traditional IPFW firewall service which had its strengths and weaknesses. The last years though have brought the project two new packet filtering services: Darren Reed's IPFilter and OpenBSD's Packet Filter.

The IPFW framework appeared in FreeBSD in the early 2.0 releases and has been extended and reworked over time. The `dummynet(4)` traffic shaping module has been introduced in FreeBSD 2.2.8 and statefull extensions have been committed around FreeBSD 4.0. In summer 2002 the IPFW engine has seen a major overhaul under the "ipfw2" codename.

The IPFilter framework is the work of Darren Reed, a packet filtering service that was designed to be portable across several Unix distributions. It offers advanced packet filtering and NAT services out of the box; some of its strengths are the packet authentication, statefull packet tracking and hierarchical rulesets.

*You can contact the author at *ady@freebsd.ady.ro* or see his webpage on *www.ady.ro*

The Packet Filter framework is an offspring of the IPFilter package; it was written by the OpenBSD folks as an alternative to Darren Reed's IPFilter when they learnt that the licensing terms were not open enough to permit uncontrolled derivative works. It has all that IPFilter offered and even a lot more; its only drawback may currently be that it's hierarchical rulesets feature doesn't allow nesting. Lately it has been adapted to work with the ALTQ traffic shaping framework with which combined they offer network QoS[1] services. Until recently the PF framework was available only as a port for FreeBSD 5.x, but this summer the framework has been imported in the FreeBSD-CURRENT branch.

1.2 Common issues

One important aspect that you always need to take into account is that the forwarded packets enter the kernel twice, first when they are received inbound and second when they leave the system outbound. This means that these packets will travel through the firewall's rules twice and you will most probably want to process them once. One workaround for this issue is using the statefull extensions; make sure though that you have enough resources to scale up to your traffic flow needs.

Firewalls have a default policy regarding packet handling when no user configured rule has been matched. This policy may be either "open", which means the packets will be permitted to travel the firewall by default, or "closed", in which case the packets will be dropped by default. This policy can be adjusted either from the kernel configuration file at compile time or at run time through a sysctl or a firewall rule. Most users will probably leave their firewall "open" for usability reasons but paranoid users will always choose to "close" it.

Be very careful when reconfiguring your firewall remotely! Do not forget that your remote session is basically a network transmission and any mistake may result in immediate and definitive isolation from the system you were reconfiguring. Remember that any command output is also sent through the network and if you loose the network connection then the command chain you launched may not be entirely processed; try redirecting the output or make the utilities run "quietly". For these reasons you should always try to do your firewall reconfiguration from the systems' console and not through a networked session.

And last, but not least, make sure you have enough network buffers – otherwise you may experience packet drop problems, usually detectable locally on the machine from the "no buffer space" errors. Tune up your NMBCLUSTERS paying attention to the `tuning(7)` manual page.

1.3 "Hello world !"

Before going deeper into the details let's have a taste of what we are offered with an example firewall configuration for each of the three frameworks.

[1] Quality of Service – a framework which permits guaranteed service provisioning

Let's suppose we have a machine with the IP 80.0.0.1 and we want to make sure that only two systems with IPs 190.0.0.1 and 190.0.0.2 may connect through SSH to our machine. We will assume that the SSH service runs at the default port 22 and no specific default firewall policy.

```
add 1000 allow tcp from 190.0.0.1,190.0.0.2 to 80.0.0.1 22 in
add 1100 allow tcp from 80.0.0.1 22 to 190.0.0.1,190.0.0.2 out
add 2000 deny tcp from any to 80.0.0.1 22
```

Figure 1: IPFW ruleset example

In figure 1 we see a sample IPFW ruleset that either needs to be loaded with "`ipfw -f <file>`" or you can load it one rule at a time by running `ipfw` with each rule as the argument. Observe that we specified the "in" and "out" keywords to exactly match the incoming and outgoing packets and that we need to catch both the incoming and outgoing packets.

```
pass  in  quick proto tcp from 190.0.0.1 to 80.0.0.1 port = 22
pass  in  quick proto tcp from 190.0.0.2 to 80.0.0.1 port = 22
pass  out quick proto tcp from 80.0.0.1 port = 22 to 190.0.0.1
pass  out quick proto tcp from 80.0.0.1 port = 22 to 190.0.0.2
block in  quick proto tcp from any to 80.0.0.1 port = 22
```

Figure 2: IPFilter ruleset example

The same configuration for the IPFilter framework is presented in figure 2 and it can be loaded with "`ipf -f <file>`". Observe that the rules are unnumbered and that the packet does not exit the firewall when matching a rule except when told so through the "quick" keyword. While the "in" and "out" keywords were optional for IPFW, they are required in the IPFilter rules. Unfortunately one cannot specify multiple distinct addresses on the same rule except when they can be aggregated into one network subnet.

```
me = "80.0.0.1"
my_hosts = "{ 190.0.0.1, 190.0.0.2 }"

pass  in  quick proto tcp from $my_hosts to $me port 22
pass  out quick proto tcp from $me port 22 to $my_hosts
block in  quick proto tcp from any to $me port 22
```

Figure 3: PF ruleset example

Looking at figure 3 you probably started wandering if are a Perl class by now, but don't worry, we are still on track with the PF framework; while the other firewall

frameworks may use macros in combination with an external pre-processor, the PF framework has internal support for them. If you replace the macro definitions you will easily recognise the same rule format from the IPFilter; however, PF holds many more surprises for us. These rules should be loaded with "`pfctl -f <file>`".

2 Statefull firewalling

2.1 Why do we need it

You saw in the previous section that building even a simple ruleset can be troublesome. One of the main problems is that we need to keep track of both the incoming and the outgoing packets. There is also a problem of scalability: when the rulesets grow very large and complex and the traffic flowing through the machine is big the time it takes for one packet to traverse the entire ruleset becomes a sensitive issue.

The statefull extension has started from a simple idea: one needs to check only the first packet of a data connection to determine whether the entire transmission will be allowed or not. If this packet is to be passed on then this connection will be marked in a dynamic firewall table where it will stay until this connection will be terminated or it will timeout.

We can already discern that this statefull extension is applicable for those protocols that can have a state recorded in time; it is the case of TCP mainly, but also UDP and ICMP with some approximations. The firewall will recognise the TCP connection states from the flags header, usually the SYN, FIN, RST and ACK flags. Some firewalls will even allow you to specify which flags may insert a new connection state.

2.2 How can it be done

Each of the three firewall frameworks has statefull firewalling extensions. We will rewrite the previous examples using these statefull extensions.

```
add 1000 check-state
add 1100 allow tcp from 190.0.0.1,190.0.0.2 to 80.0.0.1 22 setup \
                  keep-state in
add 2000 deny tcp from any to 80.0.0.1 22
```

Figure 4: IPFW statefull example

As you can see, IPFW needs a special rule to make him check the dynamic states table; the "setup" keyword will catch only packets that initiate a TCP connection. A similar mechanism exists in both IPFilter and PF but is more refined: the "flags" keyword permits specifying which flag bits need to be checked from a predefined set in order to match the rule. Commonly used flags are S(YN), A(CK) and R(ST).

```
pass  in  quick proto tcp from 190.0.0.1 to 80.0.0.1 port = 22 \
                flags S/SA keep state
pass  in  quick proto tcp from 190.0.0.2 to 80.0.0.1 port = 22 \
                flags S/SA keep state
block in  quick proto tcp from any to 80.0.0.1 port = 22
```

Figure 5: IPFilter statefull example

```
me = "80.0.0.1"
my_hosts = "{ 190.0.0.1, 190.0.0.2 }"

pass  in  quick proto tcp from $my_hosts to $me port 22 \
                flags S/SA keep state
block in  quick proto tcp from any to $me port 22
```

Figure 6: PF statefull example

Querying the dynamic tables is possible for all frameworks with commands such as "`ipfw pipe show`", "`ipfstat -sl`" and "`pfctl -s state`".

One big advantage of the statefull extensions is that the search time for packets is dramatically reduced. For example, in the PF case, the search time is reduced from an $O(n)$ search in the entire ruleset to an $O(\log_2 n)$ binary tree search in the states table.

2.3 The drawbacks

Not everything is perfect; this is true for the statefull technique too. Although it helps a lot optimizing and simplifying the firewall rulesets, it comes with a warning: watch out for resource consumption. The tables used for dynamic state tracking have a finite limit and DoS attack may be possible by flooding the firewall with connection attempts (usually SYN packets).

This is why one needs to periodically check the dynamic tables status and adjust various sysctls according to the kind of traffic that is passing through the firewall.

3 Traffic Shaping

3.1 Foreword

Please note that traffic shaping is not equivalent with QoS: the traffic shapers in general only limit or enforce network parameters, while the QoS frameworks must also guarantee these network parameters in a shared environment. For example, a traffic shaper will usually only limit bandwidth, but the QoS framework must also guarantee a minimal contracted bandwidth.

3.2 IPFW with Dummynet

The Dummynet module offers two objects which can help the administrator to shape the network parameters: the pipe and the queue. The module is not compiled in the kernel by default so you will need to recompile it with the following options:

```
options IPFIREWALL
options DUMMYNET
options HZ=1000     # not required but strongly recommended
```

Basically a pipe emulates a network link with a given bandwidth, propagation delay, queue size and packet loss rate. The queue is used in conjunction with the WF2Q+[2] policy which permits associating weights to network flows sharing the same bandwidth.

```
pipe 1 config bw 256Kbit/s
add 60000 pipe 1 ip from any to 192.168.0.0/24 out
```

Figure 7: Simple LAN clients limitation with IPFW pipes

Let's review in figure 7 a simple case in which we limit to 256Kbps the download of the LAN clients on the 192.168.0.0/24 network segment behind our firewall. As you can see we need to separately configure the pipe's emulated link characteristics and a firewall rule which will catch packets and place them in the pipe's network queue. Notice the "out" keyword which specifies that only outgoing packets will be caught; if we would have forgotten this keyword then the packets would have been processed twice and the clients would would get only half of the specified bandwidth limit.

Contrary to the default IPFW rules behaviour, once a packet matches a pipe rule it will exit the firewall; that's why these rules should be numbered close to the end of the firewall (65535 is the last "default policy" rule). This behaviour can be controlled through the *net.inet.ip.fw.one_pass* sysctl.

The Dummynet module offers another useful feature: dynamic pipes. Let's rewrite the above example considering that that each client from the 192.168.0.0/24 class should have its download limited to 64Kbps.

```
pipe 1 config bw 64Kbit/s buckets 256 mask dst-ip 0x000000ff
add 60000 pipe 1 ip from any to 192.168.0.0/24 out
```

Figure 8: Limiting per client with IPFW dynamic pipes

As you see in figure 8, each packet caught by the pipe rule is checked against the destination IP mask so that for each separate IP from the 192.168.0.X segment will have a separate dynamic rule cloned from the generic specified pipe. The "buckets" keyword

[2]Worst-case Fair Weighted Fair Queueing, an efficient variant of the WFQ policy

```
pipe 1 config bw 256Kbit/s
queue 10 config pipe 1 weight 10 mask dst-ip 0x000000ff
add 60000 queue 10 ip from any to 192.168.0.0/24 out
```

Figure 9: Weighted bandwidth sharing with IPFW queues

increases the number of hash table entries to meet the maximum possible number of separate dynamic rules.

Even further refinements can be done: the queue object helps us "connect" a set of flows to the same pipe in order to share its bandwidth proportionally to their weights. Please notice that these weights are not priorities so even a lower weight flow is guaranteed to get its bandwidth share even if other higher weighted flows have backlogged packets.

Figure 9 presents a configuration that uses dynamic queues cloned from a generic queue which are all connected to the same pipe so they share the same 256Kbps bandwidth. Because the weights are all the same each client will be given an equal share from the total bandwidth.

The configuration can be refined even further using the RED[3] queue management algorithm which will change the default "drop from the tail" policy and improves the TCP decongestion response time.

3.3 PF with ALTQ

Although harder to install and configure than Dummynet, the PF/ALTQ combo offers a superior class of service. The ALTQ[4] module offers a QoS traffic management framework with various packet scheduling algorithms (currently only CBQ[5], HFSC[6] and PRIQ[7] are supported).

FreeBSD support for the the PF and ALTQ frameworks is available only on the 5.x/6.x platforms. These frameworks have already been imported in the -CURRENT branch; if your machine is not running -CURRENT then you may use the `security/pf` port.

You will need in the kernel configuration file the options exemplified in figure 10 (the "`device pf*`" lines are not not needed if PF is loaded as a module, e.g. as installed from the `security/pf port`):

When it comes to configuring the firewall rules, the above mentioned packet schedulers need to be attached to an interface from which an hierarchical tree of classes will

[3] Random Early Detection – packets are randomly dropped when the queue is about to become full

[4] Alternate Queueing – an extensible BSD QoS framework, see `http://www.csl.sony.co.jp/person/kjc/kjc/software.html`

[5] Class Based Queueing

[6] Hierarchical Fair Service Curve

[7] Priority Queueing

```
# PF support
options PFIL_HOOKS
device pf
device pflog
device pfsync

# ALTQ support
options ALTQ
options ALTQ_CBQ
options ALTQ_HFSC
options ALTQ_PRIQ
options ALTQ_RED
#options ALTQ_NOPCC # only for SMP kernels
options HZ=1000     # not required but strongly recommended
```

Figure 10: Kernel configuration file for PF and ALTQ support

be organised. Packets will be sorted out in these classes based on normal PF rules bearing the "queue" keyword.

This mean you need to specify on which interface(s) these schedulers will be activated; traffic shaping can only be done for packets leaving the system through these interfaces. This is not always convenient but this is a basic requirement of the scheduling algorithms.

Figure 11 presents a configuration file excerpt for a typical situation: an office with an 1Mbps network connection (through the fxp0 interface) who uses web, mail and SSH services. The "default" class is used to "capture" all traffic that does not fall in other classes. The bandwidth will be shared among these four classes but each class will have a guaranteed share (70% for the web traffic, 10% for the mail traffic, 15% for SSH connections and 5% for what remains). Observe that the sum of the child class percentages need to make 100%. As a bonus, the CBQ scheduler permits that a class who has the "borrow" keyword in its definition to borrow bandwidth from the parent class if there is any available unused bandwidth left.

Each of the main classes may have other subsequent classes; it is the case of the "web" and "ssh" classes. Notice that the "acct" class may only use up to 40% the web bandwidth, while the "sales" class may use more than its 70% guaranteed bandwidth share, but only if the "acct" class is not entirely using his 40% share.

The "ssh" class is made up of two "interactive" and bulk" subclasses; they do not have a percentage share specified but priorities only which will make the scheduler to always choose to serve the interactive class first.

In the second section of the file we match the packets going out through the fxp0 interface (remember, the QoS schedulers can only shape outgoing traffic) to their respective class. Packet matching is done on a "last matched rule" base, putting by

```
sales_dept = "10.0.1.0/24"
acct_dept = "10.0.2.0/24"

altq on fxp0 cbq bandwidth 1Mb queue { default, web, mail, ssh }
queue default bandwidth 5% cbq(default)
queue web bandwidth 70% priority 5 cbq(borrow red) { sales, acct }
queue  sales bandwidth 60% cbq(borrow)
queue  acct bandwidth 40%
queue mail bandwidth 10% priority 0 cbq(borrow ecn)
queue ssh bandwidth 15% priority 3 { ssh_interactive, ssh_bulk }
queue  ssh_interactive priority 7
queue  ssh_bulk priority 0

pass out on fxp0 all queue default
pass out on fxp0 proto tcp from $sales_dept to any port 80 \
                 keep state queue sales
pass out on fxp0 proto tcp from $acct_dept to any port 80 \
                 keep state queue acct
pass out on fxp0 proto tcp from any to any port 22 \
                 keep state queue(ssh_bulk, ssh_interactive)
pass out on fxp0 proto tcp from any to any port 25 \
                 keep state queue sales
```

Figure 11: PF/ALTQ QoS example

default all packets in the "default" class.

For the SSH classes we used a tuple for the "queue" keyword where normally we should have specified a single class. The second class will be used for the matched packets which have a "low-delay" TOS[8] or they are TCP ACK packets with no data payload.

[8]Type Of Service