Global Information Assurance Certification Paper

Derek Schwenk 02/18/2004

OpenBSD 3.4 and PF: A Firewall Solution

Abstract: Why an OpenBSD firewall?

Choosing the right perimeter for your network is usually determined by a budget. The old adage “you get what you pay for” comes to mind when considering any purchase. There is a compromise of price versus performance. Luckily for network security there is a free open-source operating system in OpenBSD that focuses on security to provide a solid firewall foundation.

A firewall is a means of protecting hosts and networks from illicit traffic from other hosts and networks. Illicit traffic includes disruption of services or unauthorized access to private data. The firewall examines network traffic packet headers and payload against a set of rules. Based on these rules the packets are allowed or disallowed to the intended destination.

A firewall is not the holy grail of network security. An improper rule set could allow dangerous traffic to reach your vulnerable networks. Firewalls allowing traffic to unpatched and vulnerable services within your network are at risk. Most administrators protect their network from external users, but forget to watch their internal users. Confidential data could leak out internally either deliberately(corporate espionage) or unintentionally(virus/worm). A firewall is a great traffic cop but it can’t stop all illicit activities.

Background: OpenBSD and IPF vs. PF

OpenBSD(`http://www.openbsd.org`) is a free open-source BSD-based operating system with primary emphasis in the areas of proactive security and integrated cryptography. For example, OpenBSD was the first to ship a working implementation of IPSec. (1) The project is also closely tied to OpenSSH. The goals of striving to be the most secure operating system and maintaining a high level of hardware portability makes OpenBSD a great resource for securing small or large networks with a free software based firewall which can run on many different hardware platforms.

Taking a glance around the Internet, you’ll notice that ipfilter(`ipf`) used to be part of the OpenBSD distribution. If you install OpenBSD 3.0 or later, you’ll notice `ipf` is gone and replaced by packet filter(`pf`). What happened?

As of OpenBSD 3.0, `ipf` was removed from the OpenBSD distribution. The head developer for OpenBSD, Theo DeRaadt, decided the licensing interpretation changes made by `ipf` developer Darren Reed no longer offered `ipf` freely available which goes against a main OpenBSD goal of providing “source code that anyone can use for ANY PURPOSE, with no restrictions.” (10)

Over the past few years, `pf` has earned a solid reputation as performance, options, stability, and ease of configuration. This reputation is earned because of a wide range of options, some of which aren’t offered by commercial firewall products. (1) For the purposes of this paper, I’ll focus on the first four items in the list below. Packet filter options include:

- Packet filtering (IPv4 and IPv6)
- stateful packet inspection(SPI)

- network address translation(NAT)
- packet logging and analysis
- dynamic rulesets
- bandwidth shaping
- load balancing
- spam filtering
- user authentication

Getting Started: Prerequisites

The first step is to obtain OpenBSD 3.0 or later(currently 3.4). The recommended method is the purchase the official CD-ROM set from the OpenBSD website to support the developers. The source is also available for free download to create your own CD-ROM set. A DOS boot disk will come in handy if you need to configure hardware options, such as RAID, with custom utilities.

As for hardware platforms, most users will use Intel i386 or compatible(i.e. AMD) architecture as a simple choice. OpenBSD also supports various other hardware architectures such as Sun SPARC/UltraSPARC, Digital Alpha, and Motorola 68k/PowerPC systems. Consult `http://www.openbsd.org/plat.html` for a more complete list.

The firewall must process, and optionally log, each packet that comes across the connected networks. As you increase traffic and features(complex rules, logging, network intrusion detection systems) on the firewall, you'll need to upgrade the hardware. A PC with a relatively new CPU, 128MB RAM, 20GB hard drive and two network cards is a good starting point. The entire list of supported hardware, video and network cards for example, can be found in the `HARDWARE` file of the OpenBSD distribution's root directory.

Installation: Quick Guide

Don't connect your network card to the internet yet! Let's wait until we've hardened and configured OpenBSD before making the firewall publicly accessible. We can also test the firewall without an internet connection by using a switch/hub or crossover ethernet cable between two network cards.

Depending on the media you are using to install OpenBSD, there are two options. The first is to boot from the official OpenBSD CD-ROM. If you are using the free downloaded distribution, you'll need to create a boot disk. The `34` number in the files below denotes the OpenBSD 3.4 distribution.

To create a DOS boot disk with Windows 9x, from the OpenBSD root directory:

```
D:\openbsd> rawrite.exe floppy34.fs a:
```

To create a DOS boot disk with Windows NT/2000/XP, from the OpenBSD root directory:

```
D:\openbsd> ntrw.exe floppy34.fs a:
```

Use `floppyB34.fs` for computers with SCSI, Gigabit ethernet or RAID devices. Try `floppyC34.fs` for laptop computers for PCMCIA and Cardbus support.

Follow the installation prompts to complete the basic OS installation. See `http://www.openbsd.org/faq/faq4.html#Install` for more details on disk partitioning and custom installations.

Hardening the OS

OpenBSD is very secure with the default installation. There has only been one remote exploit in the default OpenBSD install in over seven years. (10) Research and past experience show this is exponentially more secure than popular operating systems such as Microsoft Windows and Redhat Linux. Although OpenBSD has a great track record there are still additional steps, such as patching the operating system or disabling non-essential services, which you can perform to be even more secure.

The default installation provides a high level of security because few services are enabled by default. Most unnecessary services like telnet, ftp, and finger are disabled. OpenBSD does run services as ident, daytime and time by default. Most users will not need these services and stopping them will tighten security another notch. This can be changed by editing the `/etc/inetd.conf` file as root. The file will have entries such as:

```
ident           stream  tcp     nowait  _identd /usr/libexec/identd
identd -el
ident           stream  tcp6    nowait  _identd /usr/libexec/identd
identd -el
daytime         dgram   udp     wait    root    internal
daytime         dgram   udp6    wait    root    internal
time            dgram   udp     wait    root    internal
time            dgram   udp6    wait    root    internal
```

Adding a # in front of each of these lines tell inetd not to start these services. Restart inetd by typing:

```
openbsd# kill -HUP ‘cat /var/run/inetd.pid‘
```

Staying up to date with patches is another critical part of security practices. Experience with other operating systems has taught most users to stay vigilant. Patching OpenBSD takes some preparation and additional hardware to keep the firewall secure. Most OpenBSD patches are done by compiling source code. The caveat is source code requires compilers, which are very dangerous on a firewall. The solution is to compile binaries on a different computer behind the firewall that is running the same version of OpenBSD running on the firewall. Never apply patches with different OpenBSD versions. To apply patches, you’ll need to install the compiler tools in `comp34.tgz`. Consult `http://www.openbsd.org/errata.html` for the list of patches and `http://www.openbsd.org/faq/upgrade-minfaq.html` for patching details. (1)

Another hardening option to consider is encrypting the swap file. There may be critical data written to swap that attackers could use. This isn’t critical on a firewall but can be enabled. To change this setting immediately:

```
openbsd# sysctl –w vm.swapencrypt.enable=1
```

To restore this setting at startup, open `/etc/sysctl.conf` and make sure the following line is not commented:

```
vm.swapencrypt.enable=1
```

Packet Filtering

Packet filtering is the heart of a firewall. The language which instructs the firewall what traffic logic to perform are called rules and a collection of these rules make up a ruleset. These rules apply to network traffic entering and leaving network interfaces on the firewall.

The ruleset is evaluated from top to bottom with the last matching rule taking precedent. It is very important to note that the last matched rule is performed. If you allow port 80 as the first rule then deny all traffic as the last rule, the port 80 traffic will be blocked. The ordering of rules is critical as rules become more complex. Further tuning can also be done to optimize the rule processing.

Before constructing the ruleset, I will review the basic `pf` rule syntax: `pass` and `block`. Text surrounded by `{}` symbols signify tokens to be substituted with values. The `|` symbol denotes a logical OR; e.g. pass OR block, but not both. Items in **`bold`** are keywords. The most common rule syntax is as follows:

```
pass|block in|out on {interface} proto {protocol} from {source} port
{source port} to {destination} port {destination port}
```

`pass|block`	Allow(`pass`) or deny(`block`) the traffic matched by this rule
`in|out`	traffic entering or leaving the interface; direction is relative to the firewall; packets are entering the firewall are incoming and packets are leaving the firewall outgoing
`interface`	the interface name
`proto`	the layer 4 protocol: commonly TCP, UDP or ICMP; name to protocol mappings can be found in /etc/protocols
`source`	source address or network
`source port`	source port(s); name to port mappings can be found in /etc/services
`destination`	destination address or network
`destination port`	destination port(s); name to port mappings can be found in /etc/services

Let's start with a few example rules before starting with a ruleset. The first example is to block all traffic using the `all` keyword. The `all` keyword can be used to identify and group any interface, any source or any destination. This rule blocks incoming traffic on all interfaces with all protocols from all sources to all destinations.

```
block in all
```

The `any` keyword can be used to a lesser extent to group just interfaces, addresses or ports. If the `port` keyword is not included, `pf` will assume all ports. This rule groups all sources to be allowed in on a specific interface(`xl0`) with the TCP

protocol(`proto tcp`) from any port(no `port` keyword) to a specific destination(`$webserver`) and port(`80`).

```
pass in on xl0 proto tcp from any to $webserver port 80
```

An important feature has just been introduced to the rule syntax: the macro. Just like declaring a global variable when programming, the macro is declared once with the `$` symbol. The web server address is set once and the `$webserver` macro is used to refer to that address. This makes the rules more readable, easier to maintain and less prone to errors.

Without the `any` keyword, the rule will match against the specific interface, destination and port. This rule allows traffic on a specific interface(`xl0`) with the TCP protocol(`proto tcp`) from a specific source network(`192.168.100.0/24`) and all source ports(no `port` keyword) to a specific host destination(`192.168.5.5`) and a specific port(`80`).

```
pass in on xl0 proto tcp from 192.168.100.0/24 to 192.168.5.5 port 80
```

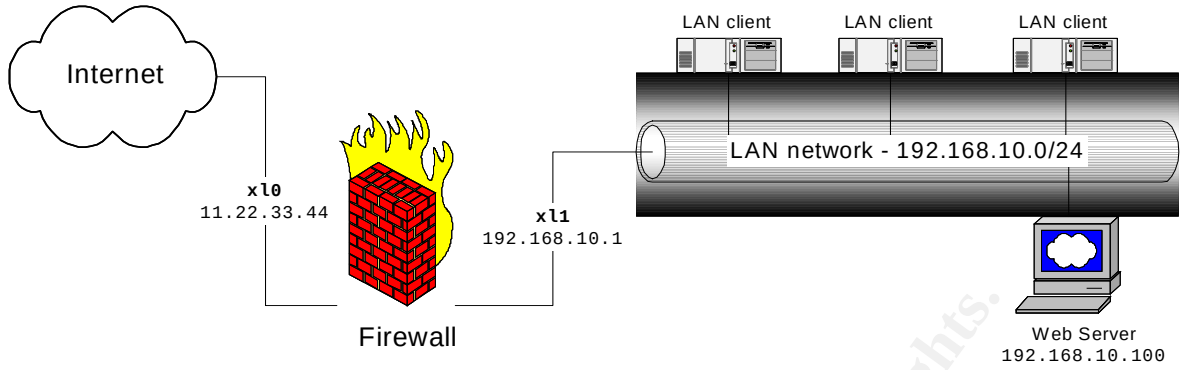
Simple Ruleset

There are two paradigms of constructing a ruleset. The first is to allow all traffic by default and block specific traffic. The second is to perform the opposite and block all traffic by default and only allow specific traffic. The latter makes sense as a more secure solution unless outside requirements force the first paradigm.

Below is a template for the example ruleset. (2) This is split into four sections: macros, global options, network address translation(NAT) and filter rules. Splitting the ruleset makes it easier to understand and maintain. Any line the starts with a `#` symbol is a comment and not processed by the `pf` engine.

```
#=== Macros ===
#=== Global Options ===
#=== NAT ===
#=== Filter Rules ===
```

The diagram below illustrates the network that will be the basis for the example ruleset. The illustration simulates an internal RFC1918 network behind a firewall with a static IP address from an ISP. The LAN will include standard PC clients requiring internet connectivity and a web server to serve customers.



Following the paradigm to block all traffic by default, let's start with the following ruleset. This ruleset isn't very functional as it blocks all incoming and outgoing traffic on each interface. Since the last matching rule is used, we can add `pass` rules later to only open access to services we want. All other traffic will be blocked by default. The last two rules allow unrestricted traffic on the localhost interface since the `block all` rule includes the localhost interface.

```
#=== Macros ===
EXT_IF="xl0"
INT_IF="xl1"
EXT_IP="11.22.33.44"
INT_IP="192.168.10.1"
INT_LAN="192.168.10.0/24"
#=== Global Options ===
#=== NAT ===
#=== Filter Rules ===
# Block everything on all interfaces
block all
# Allow unrestricted traffic on localhost
pass in on lo0 all
pass out on lo0 all
```

The macros should be edited to fit the configuration of your network. The above presumes two network cards: one for the external IP address and the other an internal RFC1918 network for LAN clients. (7)

`EXT_IF="xl0"`	manufacturer or chipset of the network card; "xl" is the 3Com 3x9xx Etherlink XL card; "0" marks the first 3Com card
`INT_IF="xl1"`	"1" marks the second 3Com card
`EXT_IP="11.22.33.44"`	External static IP address provided by your ISP
`INT_IP="192.168.10.1"`	Internal static IP acting as a gateway for LAN clients
`INT_LAN="192.168.10.0/24"`	RFC1918 network for LAN clients

Stateful Packet Inspection

At this point we have a great firewall that doesn't allow any traffic over its network interfaces. Let's add some pass rules to make the firewall useful. It's not recommended to run services on the firewall, but remote administration would be helpful. OpenSSH is installed by default with OpenBSD. An additional bonus is OpenSSH is primarily developed by the OpenBSD project. The next rule will allow ssh access on the firewall. The ssh port is mapped to the `22/tcp` entry in `/etc/services`. To use a service on a non-standard port, the port number or related macro would be entered.

```
pass in on $EXT_IF proto tcp from any to $EXT_IP port ssh keep state
```

The pass rule goes after the block rules because `pf` follows the last matched rule. The block matches first, but the pass rule is matched at the very end and the ssh traffic is allowed.

Lastly and most importantly, the "`keep state`" keyword is used to initiate stateful packet inspection for the ssh traffic. This allows `pf` to track the state of the network connection to determine if the packet belongs to an existing connection. Since the rules are blocking outgoing traffic, the ssh return traffic would not be allowed to leave the firewall. With stateful packet inspection, this traffic is allowed to leave without an additional rule. Plus the firewall will provide additional protection from spoofed traffic by attackers. This should be used for most rules, even udp and icmp traffic, to provide the extra security of stateful inspection.

When tracking state on TCP connections, the state table inspects the TCP sequence numbers in the packets. Some TCP/IP stack implementations use easily predictable initial sequence numbers making the traffic more susceptible to attackers guessing the sequence numbers with spoofed traffic. To prevent these type of attacks, use "`modulate state`" to provide a more random initial sequence number for the related rule. The additional benefit is make it more difficult for attackers to "fingerprint" the server's operating system by monitoring the behavior the TCP sequence number generation.

When using UDP and ICMP, do not apply "`modulate state`" for stateful inspection. Use "`keep state`" for non-TCP rules. The improved ssh rule would be:

```
pass in on $EXT_IF proto tcp from any to $EXT_IP port ssh modulate state
```

To further improve TCP security, the rule can also look at the TCP flags in the TCP header. For most services, the TCP `SYN` flag alone initiates the connection. With the `flags` keyword, `pf` can check which TCP flag combinations are allowed to start a connection. This can prevent certain denial of service attacks such as a TCP `SYN/FIN` attack. If the `SYN` and `FIN` flags are both set, which should not happen normally, `pf` will not match the rule and the packet wil be blocked. This is done with the following syntax:

```
flags {check}/{mask}
```

The abbreviations for the TCP flags are as follows:

- `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

The least restrictive would be to check the SYN flag looking only at the SYN and ACK flags.

```
flags S/SA
```

The most restrictive is to check the SYN flag by looking at all the flags:

```
flags S/FSRPAUEW
```

Checking all the flags is much too restrictive for normal use. Although only the SYN flag should be set when initiating a TCP handshake, some TCP implementations may use other flags such as ECE. The best compromise is to check the SYN, ACK, FIN and RST flags: (9)

```
flags S/SAFR
```

The ssh rule should now be:

```
pass in on $EXT_IF proto tcp from any to $EXT_IP port ssh flags S/SAFR
modulate state
```

But this is only half of what we need to allow ssh. This is because we are filtering both sides of the interface with the `block all` rule: `in` and `out`. It is very important to remember to add a rule for each side of the interface if you are controlling outbound traffic in addition to inbound traffic. The above rule covers the `in` rule, the missing half allowing ssh traffic is to create the `out` rule.

```
pass out on $EXT_IF proto tcp from any to $EXT_IP port ssh flags S/SAFR
modulate state
```

The ruleset now exists as:

```
#=== Macros ===
EXT_IF=”xl0”
INT_IF=”xl1”
EXT_IP=”11.22.33.44”
INT_IP=”192.168.10.1”
INT_LAN=”192.168.10.0/24”
#=== Global Options===
#=== NAT ===
#=== Filter Rules ===
```

```
# Block everything on all interfaces
block all
# Allow unrestricted traffic on localhost
pass in on lo0 all
pass out on lo0 all
# Allow ssh traffic to the firewall
pass in on $EXT_IF proto tcp from any to $EXT_IP port ssh flags S/SAFR
modulate state
pass out on $EXT_IF proto tcp from any to $EXT_IP port ssh flags S/SAFR
modulate state
```

Network Address Translation(NAT)

Now that the firewall can be accessed from the internet, its time to setup the LAN clients. The LAN clients are sitting in RFC1918 address space: private and non-routable. Network address translation(NAT) allows:

- LAN clients to access the internet via the firewall's external IP address (i.e. nat)
- forwarding specific ports on the firewall's external IP address to the LAN clients (i.e. redirection)
- creating a static map of all ports on the firewall's external IP address to a single LAN client (i.e. binat) (1)

These three items allow the previously private network to have a presence on the internet while being protected by the firewall. The most common use is NAT to allow internet connectivity to LAN clients. Redirection and binat offer the functionality of running servers(i.e. web, mail, dns) more securely behind the firewall.

The following rules will setup NAT for the LAN clients and redirection for the internal web server. The macro is setup to assign the web server IP address.

```
WEBSERVER="192.168.10.100"
```

The `nat` rule tells `pf` to allow traffic on the sourced from the LAN clients(`$INT_LAN`) to be sent out to the internet masquerading on the firewall's external interface(`$EXT_IF`) as the external IP address(`$EXT_IP`).

```
nat on $EXT_IF from $INT_LAN to any -> $EXT_IP
```

The `rdr` rule captures traffic from any source to the firewall's external IP address on port 80(`www`) and forwards the packets to the web server on port 80. The ports do not have to match for redirection to work. Forwarding the external port 80 to the internal web server port 8080 could also be done.

```
rdr on $EXT_IF proto tcp from any to $EXT_IP port www -> $WEBSERVER port
www
```

There are still some missing pieces. The rules to allow this traffic to enter must be added. The key part to remember is that NAT happens before any filtering occurs. Therefore the rules must pertain to the translated IP address, not the external IP address. First, traffic to the web server traffic must be allowed:

```
pass in on $EXT_IF proto tcp from any to $WEBSERVER port www flags S/SAFR
modulate state
pass out on $INT_IF proto tcp from any to $WEBSERVER port www flags S/SAFR
modulate state
```

Second, the LAN clients to will be allowed to access all internet TCP and UDP services. Because we're using NAT, the LAN clients' internet traffic enters the firewall on the LAN interface and exits the firewall on the external interface. We will narrow this down later to control which services clients can access on the internet.

```
pass in on $LAN_IF proto tcp from $INT_LAN to any flags S/SAFR modulate
state
pass out on $EXT_IF proto tcp from $EXT_IP to any flags S/SAFR modulate
state
pass in on $LAN_IF proto udp from $INT_LAN to any keep state
pass out on $EXT_IF proto udp from $EXT_IP to any keep state
```

The entire ruleset now looks like:

```
#=== Macros ===
# Interfaces
EXT_IF="xl0"
INT_IF="xl1"
EXT_IP="11.22.33.44"
INT_IP="192.168.10.1"
# Networks
INT_LAN="192.168.10.0/24"
# Hosts
WEBSERVER="192.168.10.100"
#=== Global Options ===
#=== NAT ===
nat on $EXT_IF from $INT_LAN to any -> $EXT_IP
rdr on $EXT_IF proto tcp from any to $EXT_IP port www -> $WEBSERVER port
www
#=== Filter Rules ===
# Block everything on all interfaces
block all
# Allow unrestricted traffic on localhost
pass in on lo0 all
pass out on lo0 all
# Allow ssh traffic to the firewall
pass in on $EXT_IF proto tcp from any to $EXT_IP port ssh flags S/SAFR
modulate state
pass out on $EXT_IF proto tcp from any to $EXT_IP port ssh flags S/SAFR
modulate state
# Allow http traffic to internal web server
pass in on $EXT_IF proto tcp from any to $WEBSERVER port www flags S/SAFR
modulate state
```

```
pass out on $INT_IF proto tcp from any to $WEBSERVER port www flags S/SAFR
modulate state
# Allow LAN clients outgoing tcp/udp traffic
pass in on $LAN_IF proto tcp from $INT_LAN to any flags S/SAFR modulate
state
pass out on $EXT_IF proto tcp from $EXT_IP to any flags S/SAFR modulate
state
pass in on $LAN_IF proto udp from $INT_LAN to any keep state
pass out on $EXT_IF proto udp from $EXT_IP to any keep state
```

Before leaving this section on NAT, let's reassess stateful packet inspection. The "`keep|modulate state`" keywords are not required for "`nat/binat/rdr`" rules as the NAT keywords automatically track state.

Optimization

At this point, we have a good example of a basic stateful firewall running ssh, providing internet access for LAN clients and shielding an internal web server. As more internal networks and services are added, the ruleset can become long and cumbersome. The longer the ruleset, the more latency the firewall would add when checking until the last matched rule. This is where the `quick` keyword can make the firewall run more efficiently.

If `pf` encounters a matching rule with the `quick` keyword, `pf` immediately stops processing the ruleset and uses the corresponding rule. Putting commonly used rules at the top of the ruleset with the `quick` keyword will greatly improve performance. In this example, the webserver rule and localhost will move to the top of the filter rules with `quick` to optimize traffic.

```
#=== Filter Rules ===
# Block everything on all interfaces
block all
# Allow unrestricted traffic on localhost
pass in quick on lo0 all
pass out quick on lo0 all
# Allow http traffic to internal web server
pass in quick on $EXT_IF proto tcp from any to $WEBSERVER port www flags
S/SAFR modulate state
pass out quick on $INT_IF proto tcp from any to $WEBSERVER port www flags
S/SAFR modulate state
# Allow ssh traffic to the firewall
pass in on $EXT_IF proto tcp from any to $EXT_IP port ssh flags S/SAFR
modulate state
pass out on $EXT_IF proto tcp from any to $EXT_IP port ssh flags S/SAFR
modulate state
# Allow LAN clients outgoing tcp/udp traffic
pass in on $LAN_IF proto tcp from $INT_LAN to any flags S/SAFR modulate
state
pass out on $EXT_IF proto tcp from $EXT_IP to any flags S/SAFR modulate
state
```

```
pass in on $LAN_IF proto udp from $INT_LAN to any keep state
pass out on $EXT_IF proto udp from $EXT_IP to any keep state
```

Fragmentation

Not every packet sent over a network is well formed. Incorrect TCP/IP stack implementations may cause anomalies. Attackers can craft fragments to exploit bugs in TCP/IP stacks or bypass firewalls and network intrusion detection systems(NIDS). (1) The solution is to reassemble these fragmented packets into well formed packets. The `scrub` keyword facilitates this action.

Since scrubbing the traffic is not a trivial CPU task, choose carefully when scrubbing all interfaces. For now, we'll just scrub the traffic entering the external interface.

```
scrub in on $EXT_IF all
```

An additional benefit of scrubbing an interface is `pf` will drop incoming packets with illegal TCP flag combinations, such as `SYN/FIN` and `SYN/RST`. Therefore we can limit the `flags` mask on a scrubbed interface from `flags S/SAFR` to `flags S/SA`. The `scrub` rule is only acting on the `in` portion of the external interface therefore only those rules will have a `flags` change. The `scrub` rule will go in the beginning of the filter section of the ruleset. The filter section now consists of:

```
#=== Filter Rules ===
# Scrub incoming external traffic
scrub in on $EXT_IF all
# Block everything on all interfaces
block all
# Allow unrestricted traffic on localhost
pass in quick on lo0 all
pass out quick on lo0 all
# Allow http traffic to internal web server
pass in quick on $EXT_IF proto tcp from any to $WEBSERVER port www flags
S/SA modulate state
pass out quick on $INT_IF proto tcp from any to $WEBSERVER port www flags
S/SAFR modulate state
# Allow ssh traffic to the firewall
pass in on $EXT_IF proto tcp from any to $EXT_IP port ssh flags S/SA
modulate state
pass out on $EXT_IF proto tcp from any to $EXT_IP port ssh flags S/SAFR
modulate state
# Allow LAN clients outgoing tcp/udp traffic
pass in on $LAN_IF proto tcp from $INT_LAN to any flags S/SA modulate
state
pass out on $EXT_IF proto tcp from $EXT_IP to any flags S/SAFR modulate
state
pass in on $LAN_IF proto udp from $INT_LAN to any keep state
pass out on $EXT_IF proto udp from $EXT_IP to any keep state
```

Stop More Spoofing

We've just reviewed how to stop attackers spoofing weak TCP initial sequence numbers and fragments to bypass the network defenses. Attackers can also spoof their source address to hide their actual source address or impersonate another address. The attacker then has an extra layer to shield them when carrying out illicit activities.

The `antispoof` keyword provides protection from spoofing in two ways. The first is to block traffic that enters externally from an address on the same network. For example, the firewall's LAN interface should never see an address entering externally from the network block it is sitting on. The LAN clients can communicate to each other on the same network block without needing to access the firewall. The LAN clients are sitting on the 192.168.10.0/24 network. If the LAN interface receives an external packet from 192.168.10.13, which is part of the network 192.168.10.0/24, then the packet is spoofed.

The second method `antispoof` adds extra protection is blocking external packets with the same address as the interface. The firewall's LAN interface, 192.168.10.1, should never receive external packets with the same source address and should be considered illegitimate. To offer extra protection from spoofing on an interface, use the following rule:

```
antispoof quick for $EXT_IF inet
antispoof quick for $INT_IF inet
```

The caveat is not to use `antispoof` on the loopback interface. Since the host sends traffic to itself on the loopback interface, the rule would block traffic on localhost.

```
#=== Filter Rules ===
# Scrub incoming external traffic
scrub in on $EXT_IF all
# Block everything on all interfaces
block all
# Allow unrestricted traffic on localhost
pass quick in on lo0 all
pass quick out on lo0 all
# Anti-spoof protection
antispoof quick for $EXT_IF inet
antispoof quick for $INT_IF inet
# Allow http traffic to internal web server
pass in quick on $EXT_IF proto tcp from any to $WEBSERVER port www flags
S/SA modulate state
pass out quick on $INT_IF proto tcp from any to $WEBSERVER port www flags
S/SAFR modulate state
# Allow ssh traffic to the firewall
pass in on $EXT_IF proto tcp from any to $EXT_IP port ssh flags S/SA
modulate state
pass out on $EXT_IF proto tcp from any to $EXT_IP port ssh flags S/SAFR
modulate state
# Allow LAN clients outgoing tcp/udp traffic
```

```
pass in on $LAN_IF proto tcp from $INT_LAN to any flags S/SA modulate
state
pass out on $EXT_IF proto tcp from $EXT_IP to any flags S/SAFR modulate
state
pass in on $LAN_IF proto udp from $INT_LAN to any keep state
pass out on $EXT_IF proto udp from $EXT_IP to any keep state
```

Outgoing LAN Traffic and Lists

It's time to revisit narrowing down the rules allowing LAN clients to access only a certain set of external services. The main benefit is to prevent users from accessing dangerous external services generated from worms, malware or trojaned software. This isn't meant as a cure all but it will hopefully slow down an outbreak within your network and the internet as a whole. Another application is to enforce a company security policy preventing users from accessing unauthorized services.

This will rule will also introduce the concept of grouping variables together, called lists, to consolidate rules. The set of services the LAN clients will be able to access externally will include: smtp, pop3, www, https, and ssh. Additional services can be found by name in the `/etc/services` file.

The rule below is an example of how lists work in `pf`. When these values are surrounded by `{}` in a rule, `pf` processes the rule looking at each item in the set. The first rule below is the same as the following five rules.

```
# Rule grouped with all 5 services
pass out on $EXT_IF proto tcp from $EXT_IP to any port {smtp, pop3, www,
http, ssh} flags S/SAFR modulate state
# Five rules combined in the rule above
pass out on $EXT_IF proto tcp from $EXT_IP to any port smtp flags S/SAFR
modulate state
pass out on $EXT_IF proto tcp from $EXT_IP to any port pop3 flags S/SAFR
modulate state
pass out on $EXT_IF proto tcp from $EXT_IP to any port www flags S/SAFR
modulate state
pass out on $EXT_IF proto tcp from $EXT_IP to any port https flags S/SAFR
modulate state
pass out on $EXT_IF proto tcp from $EXT_IP to any port ssh flags S/SAFR
modulate state
```

The TCP rules for the LAN clients will appear as:

```
pass in on $LAN_IF proto tcp from $INT_LAN to any port {smtp, pop3, www,
https, ssh} flags S/SAFR modulate state
pass out on $EXT_IF proto tcp from $EXT_IP to any port {smtp, pop3, www,
https, ssh} flags S/SAFR modulate state
```

The LAN clients should only require UDP traffic for DNS lookups.

```
pass in on $LAN_IF proto udp from $INT_LAN to any port domain keep state
pass out on $EXT_IF proto udp from $EXT_IP to any port domain keep state
```

The filter rules with the modified outgoing LAN traffic is now:

```
#=== Filter Rules ===
# Scrub incoming external traffic
scrub in on $EXT_IF all
# Block everything on all interfaces
block all
# Allow unrestricted traffic on localhost
pass quick in on lo0 all
pass quick out on lo0 all
# Anti-spoof protection
antispoof quick for $EXT_IF inet
antispoof quick for $INT_IF inet
# Allow http traffic to internal web server
pass in quick on $EXT_IF proto tcp from any to $WEBSERVER port www flags
S/SA modulate state
pass out quick on $INT_IF proto tcp from any to $WEBSERVER port www flags
S/SAFR modulate state
# Allow ssh traffic to the firewall
pass in on $EXT_IF proto tcp from any to $EXT_IP port ssh flags S/SA
modulate state
pass out on $EXT_IF proto tcp from any to $EXT_IP port ssh flags S/SAFR
modulate state
# Allow LAN clients limited outgoing tcp traffic
pass in on $LAN_IF proto tcp from $INT_LAN to any port {smtp, pop3, www,
https, ssh} flags S/SAFR modulate state
pass out on $EXT_IF proto tcp from $EXT_IP to any port {smtp, pop3, www,
https, ssh} flags S/SAFR modulate state
# Allow LAN clients limited outgoing udp traffic
pass in on $LAN_IF proto udp from $INT_LAN to any port domain keep state
pass out on $EXT_IF proto udp from $EXT_IP to any port domain keep state
```

Global Options

The global options section of the ruleset is still blank. This section is used to set options which affect how `pf` acts. The `set` options I'll cover are `loginterface`, `block-policy`, and `limit`.

The `set loginterface` option is a very useful to log the traffic statistics on an interface. This setting tracks information such as number of packets and bytes passed on the interface, state table information and other counters. This can only be set on a single interface at one time. The external interface is the preferable interface for usage statistics. The setting to enable logging the interface is:

```
set loginterface $EXT_IF
```

To view the interface statistics, use the following command:

```
openbsd# pfctl –s info
Status: Enabled for 0 days 12:07:14              Debug: None
```

```
State Table                          Total             Rate
  current entries                        5
  searches                          101318            1.3/s
  inserts                              580            0.0/s
  removals                             575            0.0/s
Counters
  match                               3854            0.1/s
  bad-offset                             0            0.0/s
  fragment                               0            0.0/s
  short                                  0            0.0/s
  normalize                              0            0.0/s
  memory                                 0            0.0/s
```

The `set block-policy` option is used to as a default for rules that don't explicitly set how to handle the block rules. The `block-policy` options are `drop` and `return`. The default behavior for a block rule is `drop`. This is recommended to conserve resources and "silent" firewalls are more difficult to scan and fingerprint.

`drop`	blocked packets are silently dropped
`return`	TCP RST packets are sent for blocked TCP traffic; ICMP Unreachable packets are sent for all other blocked traffic

We'll setup the firewall to drop packets for rules that aren't specifically matched. This is redundant with the block rules I have used so far, but for example I'll use:

```
set block-policy block
```

The `set limit` option helps fine-tune the resources on the firewall. This controls the number of fragments and states kept in memory. The `frags` option sets the number of entries kept in memory when using `scrub` rules. The default is 5000 entries. The `states` option sets the amount of entries in the state table when using "`keep|modulate state`." The default is 10000 entries. If there was a firewall with additional processing power to hold more entries, the following rules could be used:

```
set limit frags 7500
set limit states 15000
```

The global options section now contains:

```
#=== Global Options ===
set loginterface $EXT_IF
set block-policy block
set limit frags 7500
set limit states 15000
```

Packet Logging and Analysis

Once the ruleset is in place, its helpful to have some logs of what `pf` is doing whether it is for troubleshooting, monitoring or forensics. Packet logging is always

enabled, but pf will only log packets when a rule contains the log or log-all keyword. Any rule can be logged. This allows you to fine tune the logs to fit your requirements.

To log when the antispoof rule is matched, the rules become:

```
antispoof log quick for $EXT_IF inet
antispoof log quick for $INT_IF inet
```

The pf logs are stored in binary format so you can't view them with a text editor. A tool such as tcpdump can facilitate this. Normal tcpdump flags can be used to filter the log. The packets marked to be logged appear in /var/log/pflog. To view with tcpdump:

```
/usr/sbin/tcpdump –r /var/log/pflog
```

Older logs may have been archived into gzip format to save disk space. The archived files will appear as /var/log/pflog{1-3}.gz. Unarchive the .gz files with gunzip before using tcpdump.

Viewing the log file with tcpdump will not provide real-time data. To troubleshoot pf in real-time, tcpdump should be used on the pflog0 interface. The pflog0 interface copies all the packets pf is configured to log.

```
/usr/sbin/tcpdump –i pflog0
```

Starting PF

Now that we have a complete ruleset we're ready to start pf. The ruleset above resides in the /etc/pf.conf file; the default configuration file for pf. Before starting pf manually, let's ensure it launches at startup after a reboot. Open /etc/rc.conf and check the two following lines exist:

```
pf=YES
pf_rules=/etc/pf.conf
```

The pf variable should be set to YES instead of NO. If you are going to use a different rules file, enter if after the pf_rules variable. Make sure the new rules file is owned by root and belongs to the wheel group. Only the root user should have read/write privileges.

```
-rw-------  1 root  wheel  5091 Jan 31  20:10 /etc/pf.conf
```

To control pf via the command line, the pfctl command is used. Starting and loading the /etc/pf.conf ruleset is done by:

```
openbsd# pfctl –f /etc/pf.conf
```

To load only the filter rules in the configuration file:

```
openbsd# pfctl –Rf /etc/pf.conf
```

To load only the NAT rules in the configuration file:

```
openbsd# pfctl –Nf /etc/pf.conf
```

To view only the current filter rules `pf` has loaded:

```
openbsd# pfctl –s rules
```

To view only the current NAT rules `pf` has loaded:

```
openbsd# pfctl –s nat
```

To view everything `pftcl` can display:

```
openbsd# pfctl –s all
```

Testing

After loading the final ruleset with `pfctl` the firewall is ready to be tested. The external interface can be connected to your internet connection or connected privately via a switch or hub for internal testing. The latter is the best solution to make sure the ruleset is safe before exposing the firewall to the public. If this is your first time working with a firewall, you may be amazed at the number of network scans.

Tools such as nmap(`www.insecure.org/nmap`) and hping(`www.hping.org`) are great to automate the testing of the firewall. Always double check if any authorization is required before doing a port scan on a firewall. Better safe than sorry.

Conclusion

Anyone connected to the internet without firewall functionality is opening the door for trouble. Certain services and data on your network are not meant to be used by people outside your organization for confidential and/or security reasons. Those services you do provide to the public should be protected from malicious intent. A firewall isn't going to cure every vulnerability or attack vector within your network, but a strong perimeter will provide a barrier from the majority of attacks.

Using a firewall can provide extra security, but a poorly configured firewall is the same as not having a firewall in the first place. Using a proper configuration with stateful packet inspection and network address translation, it is possible to have a functional network while shielding the majority of illicit traffic from the internet and within your private network.

Best of all, the foundation for the firewall is OpenBSD: a free open-source product proactively focused on security with a long track record. Whenever budgets are involved, not much sounds better than the word "free". Even $40 to support the developers shouldn't add much to the bottom line.

Appendix: Final Ruleset

```
#=== Macros ===
# Interfaces
EXT_IF=”xl0”
INT_IF=”xl1”
EXT_IP=”11.22.33.44”
INT_IP=”192.168.10.1”
# Networks
INT_LAN=”192.168.10.0/24”
# Hosts
WEBSERVER=”192.168.10.100”
#=== Global Options ===
set loginterface $EXT_IF
set block-policy block
set limit frags 7500
set limit states 15000
#=== NAT ===
nat on $EXT_IF from $INT_LAN to any -> $EXT_IP
rdr on $EXT_IF proto tcp from any to $EXT_IP port $HTTP -> $WEBSERVER port
www
#=== Filter Rules ===
# Scrub incoming external traffic
scrub in on $EXT_IF all
# Block everything on all interfaces
block all
# Allow unrestricted traffic on localhost
pass quick in on lo0 all
pass quick out on lo0 all
# Anti-spoof protection
antispoof quick log for $EXT_IF inet
antispoof quick log for $INT_IF inet
# Allow http traffic to internal web server
pass in quick on $EXT_IF proto tcp from any to $WEBSERVER port www flags
S/SA modulate state
pass out quick on $INT_IF proto tcp from any to $WEBSERVER port www flags
S/SAFR modulate state
# Allow ssh traffic to the firewall
pass in on $EXT_IF proto tcp from any to $EXT_IP port ssh flags S/SA
modulate state
pass out on $EXT_IF proto tcp from any to $EXT_IP port ssh flags S/SAFR
modulate state
# Allow LAN clients limited outgoing tcp traffic
pass in on $LAN_IF proto tcp from $INT_LAN to any port {smtp, pop3, www,
https, ssh} flags S/SAFR modulate state
pass out on $EXT_IF proto tcp from $EXT_IP to any port {smtp, pop3, www,
https, ssh} flags S/SAFR modulate state
```

```
# Allow LAN clients limited outgoing udp traffic
pass in on $LAN_IF proto udp from $INT_LAN to any port domain keep state
pass out on $EXT_IF proto udp from $EXT_IP to any port domain keep state
```

References

1. Artymiak, Jacek. “Building Firewalls with OpenBSD and PF”; Second Ed. devGuide.net. Poland, 2003. p322
2. Bullen, Eric. “A Newbie's Guide to Setting up PF on OpenBSD 3.x “ URL: http://www.thedeepsky.com/howto/newbie_pf_guide.php (09/17/2003)
3. Fnordia.org web site. “An OpenBSD 3.3 pf firewall” URL: http://www.fnordia.org/docs/pf.php (06/02/2003)
4. Hartmeier, Daniel. “OpenBSD Packet Filter” URL: http://www.benzedrine.cx/pf.html (01/31/2004)
5. King, Andrew. “Andrew’s Guide to OpenBSD/i386” URL: http://www.andrewsworld.org/docs/openbsd.htm (01/31/2004)
6. Matulkis, Peter. “Understand Packet Filter” URL: http://www.aei.ca/~pmatulis/pub/obsd_pf.html (2004)
7. Network Working Group. “Address Allocation for Private Networks” URL: http://www.faqs.org/rfcs/rfc1918.html (Feb 1996)
8. OpenBSD web site. “Manual Pages pf.conf” URL: http://www.openbsd.org/cgi-bin/man.cgi?query=pf.conf&sektion=5&arch=&apropos=0&manpath=OpenBSD+3.4 (11/19/2002)
9. OpenBSD web site. “PF: The OpenBSD Packet Filter” URL: http://www.openbsd.org/faq/pf/ (01/01/2004)
10. OpenBSD web site. “Project Goals” URL: http://www.openbsd.org/goals.html (08/04/2003)

Last Updated: July 24th, 2017

Upcoming Training

SANS Prague 2017	Prague, Czech Republic	Aug 07, 2017 - Aug 12, 2017	Live Event
SANS Boston 2017	Boston, MA	Aug 07, 2017 - Aug 12, 2017	Live Event
Community SANS Omaha SEC401*	Omaha, NE	Aug 14, 2017 - Aug 19, 2017	Community SANS
SANS New York City 2017	New York City, NY	Aug 14, 2017 - Aug 19, 2017	Live Event
SANS Salt Lake City 2017	Salt Lake City, UT	Aug 14, 2017 - Aug 19, 2017	Live Event
SANS Chicago 2017	Chicago, IL	Aug 21, 2017 - Aug 26, 2017	Live Event
SANS Virginia Beach 2017	Virginia Beach, VA	Aug 21, 2017 - Sep 01, 2017	Live Event
SANS Adelaide 2017	Adelaide, Australia	Aug 21, 2017 - Aug 26, 2017	Live Event
Community SANS Trenton SEC401	Trenton, NJ	Aug 21, 2017 - Aug 26, 2017	Community SANS
Virginia Beach 2017 - SEC401: Security Essentials Bootcamp Style	Virginia Beach, VA	Aug 21, 2017 - Aug 26, 2017	vLive
Community SANS Pasadena SEC401 @ NASA	Pasadena, CA	Aug 23, 2017 - Aug 30, 2017	Community SANS
Mentor Session - SEC401	Minneapolis, MN	Aug 29, 2017 - Oct 10, 2017	Mentor
SANS San Francisco Fall 2017	San Francisco, CA	Sep 05, 2017 - Sep 10, 2017	Live Event
SANS Tampa - Clearwater 2017	Clearwater, FL	Sep 05, 2017 - Sep 10, 2017	Live Event
Mentor Session - SEC401	Edmonton, AB	Sep 06, 2017 - Oct 18, 2017	Mentor
SANS Network Security 2017	Las Vegas, NV	Sep 10, 2017 - Sep 17, 2017	Live Event
Community SANS Albany SEC401	Albany, NY	Sep 11, 2017 - Sep 16, 2017	Community SANS
Mentor Session - SEC401	Ventura, CA	Sep 11, 2017 - Oct 12, 2017	Mentor
Community SANS Columbia SEC401	Columbia, MD	Sep 18, 2017 - Sep 23, 2017	Community SANS
Community SANS Dallas SEC401	Dallas, TX	Sep 18, 2017 - Sep 23, 2017	Community SANS
Community SANS Boise SEC401	Boise, ID	Sep 25, 2017 - Sep 30, 2017	Community SANS
Baltimore Fall 2017 - SEC401: Security Essentials Bootcamp Style	Baltimore, MD	Sep 25, 2017 - Sep 30, 2017	vLive
Community SANS New York SEC401	New York, NY	Sep 25, 2017 - Sep 30, 2017	Community SANS
Rocky Mountain Fall 2017	Denver, CO	Sep 25, 2017 - Sep 30, 2017	Live Event
SANS London September 2017	London, United Kingdom	Sep 25, 2017 - Sep 30, 2017	Live Event
SANS Baltimore Fall 2017	Baltimore, MD	Sep 25, 2017 - Sep 30, 2017	Live Event
SANS Copenhagen 2017	Copenhagen, Denmark	Sep 25, 2017 - Sep 30, 2017	Live Event
Community SANS Sacramento SEC401	Sacramento, CA	Oct 02, 2017 - Oct 07, 2017	Community SANS
SANS DFIR Prague 2017	Prague, Czech Republic	Oct 02, 2017 - Oct 08, 2017	Live Event
Community SANS Charleston SEC401	Charleston, SC	Oct 02, 2017 - Oct 07, 2017	Community SANS
Mentor Session - SEC401	Arlington, VA	Oct 04, 2017 - Nov 15, 2017	Mentor