2

PF CONFIGURATION BASICS

In this chapter, we'll create a very simple setup with PF. We'll begin with the simplest configuration possible: a single machine configured to communicate with a single network. This network could very well be the Internet.

Your two main tools for configuring PF are your favorite text editor and the `pfctl` command-line administration tool. PF configurations, usually stored in */etc/pf.conf*, are called *rule sets* because each line in the configuration file is a *rule* that helps determine what the packet-filtering subsystem should do with the network traffic it sees. In ordinary, day-to-day administration, you edit your configuration in the */etc/pf.conf* file and then load your changes using `pfctl`. There are Web interfaces for PF administration tasks, but they're not part of the base system. The PF developers aren't hostile toward these options, but they've yet to see a graphical interface for configuring PF that's clearly preferable to editing *pf.conf* and using `pfctl` commands.

The First Step: Enabling PF

Before you can get started on the fun parts of shaping your network with PF and related tools, you need to make sure that PF is available and enabled. The details depend on your specific operating system: OpenBSD, FreeBSD, or NetBSD. Check your setup by following the instructions for your operating system and then move on to "A Simple PF Rule Set: A Single, Stand-Alone Machine" on page 16.

The `pfctl` command is a program that requires higher privilege than the default for ordinary users. In the rest of this book, you'll see commands that require extra privilege prefixed with `sudo`. If you haven't started using `sudo` yet, you should. `sudo` is in the base system on OpenBSD. On FreeBSD, DragonFly BSD, and NetBSD, it's within easy reach via the ports system or pkgsrc system, respectively, as *security/sudo*.

Here are a couple general notes regarding using `pfctl`:

- The command to disable PF is `pfctl -d`. Once you've entered that command, all PF-based filtering that may have been in place will be disabled, and all traffic will be allowed to pass.
- For convenience, `pfctl` can handle several operations on a single command line. To enable PF and load the rule set in a single command, enter the following:

  ```
  $ sudo pfctl -ef /etc/pf.conf
  ```

Setting Up PF on OpenBSD

In OpenBSD 4.6 and later, you don't need to enable PF because it's enabled by default with a minimal configuration in place.[1] If you were watching the system console closely while the system was starting up, you may have noticed the `pf enabled` message appear soon after the kernel messages completed.

If you didn't see the `pf enabled` message on the console at startup, you have several options to check that PF is indeed enabled. One simple way to check is to enter the command you would otherwise use to enable PF from the command line:

```
$ sudo pfctl -e
```

If PF is already enabled, the system responds with this message:

```
pfctl: pf already enabled
```

1. If you're setting up your first PF configuration on an OpenBSD version earlier than this, the best advice is to upgrade to the most recent stable version. If for some reason you must stay with the older version, it might be useful to consult the first edition of this book as well as the man pages and other documentation for the specific version you're using.

If PF isn't enabled, the `pfctl -e` command will enable PF and display this:

```
pf enabled
```

In versions prior to OpenBSD 4.6, PF wasn't enabled by default. You can override the default by editing your */etc/rc.conf.local* file (or creating the file, if it doesn't exist). Although it isn't necessary on recent OpenBSD versions, it doesn't hurt to add this line to your */etc/rc.conf.local* file:

```
pf=YES                # enable PF
```

If you take a look at the */etc/pf.conf* file in a fresh OpenBSD installation, you get your first exposure to a working rule set.

The default OpenBSD *pf.conf* file starts off with a `set skip on lo` rule to make sure traffic on the loopback interface group isn't filtered in any way. The next active line is a simple `pass` default to let your network traffic pass by default. Finally, an explicit `block` rule blocks remote X11 traffic to your machine.

As you probably noticed, the default *pf.conf* file also contains a few comment lines starting with a hash mark (`#`). In those comments, you'll find suggested rules that hint at useful configurations, such as FTP passthrough via `ftp-proxy` (see Chapter 3) and `spamd`, the OpenBSD spam-deferral daemon (see Chapter 6). These items are potentially useful in various real-world scenarios, but because they may not be relevant in all configurations, they are commented out in the file by default.

If you look for PF-related settings in your */etc/rc.conf* file, you'll find the setting `pf_rules=`. In principle, this lets you specify that your configuration is in a file other than the default */etc/pf.conf*. However, changing this setting is probably not worth the trouble. Using the default setting lets you take advantage of a number of automatic housekeeping features, such as automatic nightly backup of your configuration to */var/backups*.

On OpenBSD, the */etc/rc* script has a built-in mechanism to help you out if you reboot with either no *pf.conf* file or one that contains an invalid rule set. Before enabling any network interfaces, the *rc* script loads a rule set that allows a few basic services: SSH from anywhere, basic name resolution, and NFS mounts. This allows you to log in and correct any errors in your rule set, load the corrected rule set, and then go on working from there.

Setting Up PF on FreeBSD

Good code travels well, and FreeBSD users will tell you that good code from elsewhere tends to find its way into FreeBSD sooner or later. PF is no exception, and from FreeBSD 5.2.1 and the 4.*x* series onward, PF and related tools became part of FreeBSD.

If you read through the previous section on setting up PF on OpenBSD, you saw that on OpenBSD, PF is enabled by default. That isn't the case on FreeBSD, where PF is one of three possible packet-filtering options. Here, you need to take explicit steps to enable PF, and compared to OpenBSD, it seems that you need a little more magic in your */etc/rc.conf*. A look at your */etc/defaults/rc.conf* file shows that the FreeBSD default values for PF-related settings are as follows:

```
pf_enable="NO"                 # Set to YES to enable packet filter (PF)
pf_rules="/etc/pf.conf"        # rules definition file for PF
pf_program="/sbin/pfctl"       # where pfctl lives
pf_flags=""                    # additional flags for pfctl
pflog_enable="NO"              # set to YES to enable packet filter logging
pflog_logfile="/var/log/pflog" # where pflogd should store the logfile
pflog_program="/sbin/pflogd"   # where pflogd lives
pflog_flags=""                 # additional flags for pflogd
pfsync_enable="NO"             # expose pf state to other hosts for syncing
pfsync_syncdev=""              # interface for pfsync to work through
pfsync_ifconfig=""             # additional options to ifconfig(8) for pfsync
```

Fortunately, you can safely ignore most of these—at least for now. The following are the only options that you need to add to your */etc/rc.conf* configuration:

```
pf_enable="YES"                 # Enable PF (load module if required)
pflog_enable="YES"              # start pflogd(8)
```

There are some differences between FreeBSD releases with respect to PF. Refer to the *FreeBSD Handbook* available from *http://www.freebsd.org/* —specifically the PF section of the "Firewalls" chapter—to see which information applies in your case. The PF code in FreeBSD 9 and 10 is equivalent to the code in OpenBSD 4.5 with some bug fixes. The instructions in this book assume that you're running FreeBSD 9.0 or newer.

On FreeBSD, PF is compiled as a kernel-loadable module by default. If your FreeBSD setup runs with a GENERIC kernel, you should be able to start PF with the following:

```
$ sudo kldload pf
$ sudo pfctl -e
```

Assuming you have put the lines just mentioned in your */etc/rc.conf* and created an */etc/pf.conf* file, you could also use the PF *rc* script to run PF. The following enables PF:

```
$ sudo /etc/rc.d/pf start
```

And this disables the packet filter:

```
$ sudo /etc/rc.d/pf stop
```

NOTE *On FreeBSD, the* /etc/rc.d/pf *script requires at least a line in* /etc/rc.conf *that reads* `pf_enable="YES"` *and a valid* /etc/pf.conf *file. If either of these requirements isn't met, the script will exit with an error message. There is no* /etc/pf.conf *file in a default FreeBSD installation, so you'll need to create one before you reboot the system with PF enabled. For our purposes, creating an empty* /etc/pf.conf *with* `touch` *will do, but you could also work from a copy of the* /usr/share/examples/pf/pf.conf *file supplied by the system.*

The supplied sample file */usr/share/examples/pf/pf.conf* contains no active settings. It has only comment lines starting with a `#` character and commented-out rules, but it does give you a preview of what a working rule set will look like. For example, if you remove the `#` sign before the line that says `set skip on lo` to uncomment the line and then save the file as your */etc/pf.conf*, your loopback interface group will not be filtered once you enable PF and load the rule set. However, even if PF is enabled on your FreeBSD system, we haven't gotten around to writing an actual rule set, so PF isn't doing much of anything and all packets will pass.

As of this writing (August 2014), the FreeBSD *rc* scripts don't set up a default rule set as a fallback if the configuration read from */etc/pf.conf* fails to load. This means that enabling PF with no rule set or with *pf.conf* content that is syntactically invalid will leave the packet filter enabled with a default `pass all` rule set.

Setting Up PF on NetBSD

On NetBSD 2.0, PF became available as a loadable kernel module that could be installed via packages (*security/pflkm*) or compiled into a static kernel configuration. In NetBSD 3.0 and later, PF is part of the base system. On NetBSD, PF is one of several possible packet-filtering systems, and you need to take explicit action to enable it.

Some details of PF configuration have changed between NetBSD releases. This book assumes you are using NetBSD 6.0 or later.[2]

To use the loadable PF module for NetBSD, add the following lines to your */etc/rc.conf* to enable loadable kernel modules, PF, and the PF log interface, respectively.

```
lkm="YES" # do load kernel modules
pf=YES
pflogd=YES
```

To load the *pf* module manually and enable PF, enter this:

```
$ sudo modload /usr/lkm/pf.o
$ sudo pfctl -e
```

2. For instructions on using PF in earlier releases, see the documentation for your release and look up supporting literature listed in Appendix A of this book.

Alternatively, you can run the *rc.d* scripts to enable PF and logging, as follows:

```
$ sudo /etc/rc.d/pf start
$ sudo /etc/rc.d/pflogd start
```

To load the module automatically at startup, add the following line to */etc/lkm.conf*:

```
/usr/lkm/pf.o - - - - BEFORENET
```

If your */usr* filesystem is on a separate partition, add this line to your */etc/rc.conf*:

```
critical_filesystems_local="${critical_filesystems_local} /usr"
```

If there are no errors at this point, you have enabled PF on your system, and you're ready to move on to creating a complete configuration.

The supplied */etc/pf.conf* file contains no active settings; it has only comment lines starting with a hash mark (`#`) and commented-out rules. However, it does give you a preview of what a working rule set will look like. For example, if you remove the hash mark before the line that says `set skip on lo` to uncomment it and then save the file, your loopback interface will not be filtered once you enable PF and load the rule set. However, even if PF is enabled on your NetBSD system, we haven't gotten around to writing an actual rule set, so PF isn't doing much of anything but passing packets.

NetBSD implements a default or fallback rule set via the file */etc/defaults/pf.boot.conf*. This rule set is intended only to let your system complete its boot process in case the */etc/pf.conf* file doesn't exist or contains an invalid rule set. You can override the default rules by putting your own customizations in */etc/pf.boot.conf*.

A Simple PF Rule Set: A Single, Stand-Alone Machine

Mainly to have a common, minimal baseline, we will start building rule sets from the simplest possible configuration.

A Minimal Rule Set

The simplest possible PF setup is on a single machine that will not run any services and talks to only one network, which may be the Internet.

We'll begin with an */etc/pf.conf* file that looks like this:

```
block in all
pass out all keep state
```

This rule set denies all incoming traffic, allows traffic we send, and retains state information on our connections. PF reads rules from top to bottom; the *last* rule in a rule set that matches a packet or connection is the one that is applied.

Here, any connection coming into our system from anywhere else will match the `block in all` rule. Even with this tentative result, the rule evaluation will continue to the next rule (`pass out all keep state`), but the traffic will not even match the first criterion (the `out` direction) in this rule. With no more rules to evaluate, the status will not change, and the traffic will be blocked. In a similar manner, any connection initiated from the machine with this rule set will not match the first rule (once again, the wrong direction) but will match the second rule, which is a `pass` rule, and the connection is allowed to pass.

We'll examine the way that PF evaluates rules and how ordering matters in a bit more detail in Chapter 3, in the context of a slightly longer rule set.

For any rule that has a `keep state` part, PF keeps information about the connection, including various counters and sequence numbers, as an entry in the *state table*. The state table is where PF keeps information about existing connections that have already matched a rule, and new packets that arrive are compared to existing state table entries to find a match first. Only when a packet doesn't match any existing state will PF move on to a full *rule set evaluation*, checking whether the packet matches a rule in the loaded rule set. We can also instruct PF to act on state information in various ways, but in a simple case like this, our main goal is to allow return traffic for connections we initiate to return to us.

Note that on OpenBSD 4.1 and later, the default for `pass` rules is to keep state information,[3] and we no longer need to specify `keep state` explicitly in a simple case like this. This means the rule set could be written like this:

```
# minimal rule set, OpenBSD 4.1 onward keeps state by default
block in all
pass out all
```

In fact, you could even leave out the `all` keyword here if you like.

The other BSDs have mostly caught up with this change by now, and for the rest of this book, we'll stick to the newer rules, with an occasional reminder in case you are using an older system.

It goes pretty much without saying that passing all traffic generated by a specific host implies that the host in question is, in fact, trustworthy. This is something you do only if this is a machine you know you can trust.

3. In fact, the new default corresponds to `keep state flags S/SA`, ensuring that only initial SYN packets during connection setup create state, eliminating some puzzling error scenarios. To filter statelessly, you can specify `no state` for the rules where you don't want to record or keep state information. On FreeBSD, OpenBSD 4.1–equivalent PF code was merged into version 7.0. If you're using a PF version old enough that it does not have this default, it is a very strong indicator that you should consider upgrading your operating system as soon as feasible.

When you're ready to use this rule set, load it with the following:

```
$ sudo pfctl -ef /etc/pf.conf
```

The rule set should load without any error messages or warnings. On all but the slowest systems, you should be returned to the `$` prompt immediately.

Testing the Rule Set

It's always a good idea to test your rule sets to make sure they work as expected. Proper testing will become essential once you move on to more complicated configurations.

To test the simple rule set, see whether it can perform domain name resolution. For example, you could see whether `$ host nostarch.com` returns information, such as the IP address of the host *nostarch.com* and the hostnames of that domain's mail exchangers. Or just see whether you can surf the Web. If you can connect to external websites by name, the rule set allows your system to perform domain name resolution. Basically, any service you try to access from your own system should work, and any service you try to access on your system from another machine should produce a `Connection refused` message.

Slightly Stricter: Using Lists and Macros for Readability

The rule set in the previous section is an extremely simple one—probably too simplistic for practical use. But it's a useful starting point to build from to create a slightly more structured and complete setup. We'll start by denying all services and protocols and then allow only those we know that we need,[4] using lists and macros for better readability and control.

A *list* is simply two or more objects of the same type that you can refer to in a rule set, such as this:

```
pass proto tcp to port { 22 80 443 }
```

Here, `{ 22 80 443 }` is a list.

A *macro* is a pure readability tool. If you have objects that you'll refer to more than once in your configuration, such as an IP address for an important host, it could be useful to define a macro instead. For example, you might define this macro early in your rule set:

```
external_mail = 192.0.2.12
```

4. Why write the rule set to default deny? The short answer is that it gives you better control. The point of packet filtering is to take control, not to play catch-up with what the bad guys do. Marcus Ranum has written a very entertaining and informative article about this called "The Six Dumbest Ideas in Computer Security" (*http://www.ranum.com/security/computer_security/editorials/dumb/index.html*).

Then you could refer to that host as $external_mail later in the rule set:

```
pass proto tcp to $external_mail port 25
```

These two techniques have great potential for keeping your rule sets readable, and as such, they are important factors that contribute to the overall goal of keeping you in control of your network.

A Stricter Baseline Rule Set

Up to this point, we've been rather permissive with regard to any traffic we generate ourselves. A permissive rule set can be very useful while we check that basic connectivity is in place or we check whether filtering is part of a problem we're seeing. Once the "Do we have connectivity?" phase is over, it's time to start tightening up to create a baseline that keeps us in control.

To begin, add the following rule to */etc/pf.conf*:

```
block all
```

This rule is completely restrictive and will block all traffic in all directions. This is the initial baseline filtering rule that we'll use in all complete rule sets over the next few chapters. We basically start from zero, with a configuration where *nothing* is allowed to pass. Later on, we'll add rules that cut our traffic more slack, but we'll do so incrementally and in a way that keeps us firmly in control.

Next, we'll define a few macros for later use in the rule set:

```
tcp_services = "{ ssh, smtp, domain, www, pop3, auth, https, pop3s }"
udp_services = "{ domain }"
```

Here, you can see how the combination of lists and macros can be turned to our advantage. Macros can be lists, and as demonstrated in the example, PF understands rules that use the names of services as well as port numbers, as listed in your */etc/services* file. We'll take care to use all these elements and some further readability tricks as we tackle complex situations that require more elaborate rule sets.

Having defined these macros, we can use them in our rules, which we'll now edit slightly to look like this:

```
block all
pass out proto tcp to port $tcp_services
pass proto udp to port $udp_services
```

The strings $tcp_services and $udp_services are macro references. Macros that appear in a rule set are expanded in place when the rule set loads, and the running rule set will have the full lists inserted where the macros are referenced. Depending on the exact nature of the macros, they may cause single rules with macro references to expand into several rules. Even in a small rule set like this, the use of macros makes the rules easier to grasp and maintain. The amount of information that needs to appear in the rule set shrinks, and with sensible macro names, the logic becomes clearer. To follow the logic in a typical rule set, more often than not, we don't need to see full lists of IP addresses or port numbers in place of every macro reference.

From a practical rule set maintenance perspective, it's important to keep in mind which services to allow on which protocol in order to keep a comfortably tight regime. Keeping separate lists of allowed services according to protocol is likely to be useful in keeping your rule set both functional and readable.

TCP VS. UDP

We've taken care to separate out UDP services from TCP services. Several services run primarily on well-known port numbers on either TCP or UDP, and a few alternate between using TCP and UDP according to specific conditions.

The two protocols are quite different in several respects. TCP is connection oriented and reliable, a perfect candidate for stateful filtering. In contrast, UDP is stateless and connectionless, but PF creates and maintains data equivalent to state information for UDP traffic in order to ensure UDP return traffic is allowed back if it matches an existing state.

One common example where state information for UDP is useful is when handling name resolution. When you ask a name server to resolve a domain name to an IP address or to resolve an IP address back to a hostname, it's reasonable to assume that you want to receive the answer. Retaining state information, or the functional equivalent about your UDP traffic, makes this possible.

Reloading the Rule Set and Looking for Errors

After we've changed our *pf.conf* file, we need to load the new rules as follows:

```
$ sudo pfctl -f /etc/pf.conf
```

If there are no syntax errors, pfctl shouldn't display any messages during the rule load.

If you prefer to display verbose output, use the -v flag:

```
$ sudo pfctl -vf /etc/pf.conf
```

When you use verbose mode, pfctl should expand your macros into their separate rules before returning you to the command-line prompt, as follows:

```
$ sudo pfctl -vf /etc/pf.conf
tcp_services = "{ ssh, smtp, domain, www, pop3, auth, https, pop3s }"
udp_services = "{ domain }"
block drop all
pass out proto tcp from any to any port = ssh flags S/SA keep state
pass out proto tcp from any to any port = smtp flags S/SA keep state
pass out proto tcp from any to any port = domain flags S/SA keep state
pass out proto tcp from any to any port = www flags S/SA keep state
pass out proto tcp from any to any port = pop3 flags S/SA keep state
pass out proto tcp from any to any port = auth flags S/SA keep state
pass out proto tcp from any to any port = https flags S/SA keep state
pass out proto tcp from any to any port = pop3s flags S/SA keep state
pass proto udp from any to any port = domain keep state
$ _
```

Compare this output to the content of the */etc/pf.conf* file you actually wrote. Our single TCP services rule is expanded into eight different ones: one for each service in the list. The single UDP rule takes care of only one service, and it expands from what we wrote to include the default options. Notice that the rules are displayed in full, with default values such as `flags S/SA keep state` applied in place of any options you do not specify explicitly. This is the configuration as it's actually loaded.

Checking Your Rules

If you've made extensive changes to your rule set, check them before attempting to load the rule set by using the following:

```
$ pfctl -nf /etc/pf.conf
```

The `-n` option tells PF to parse the rules only, without loading them—more or less as a dry run and to allow you to review and correct any errors. If `pfctl` finds any syntax errors in your rule set, it'll exit with an error message that points to the line number where the error occurred.

Some firewall guides advise you to make sure that your old configuration is truly gone, or you'll run into trouble—your firewall might be in some kind of intermediate state that doesn't match either the before or after state. That's simply not true when you're using PF. The last valid rule set loaded is active until you either disable PF or load a new rule set. `pfctl` checks the syntax and then loads your new rule set completely before switching over to the new one. This is often referred to as *atomic rule set load* and means that once a valid rule set has been loaded, there's no intermediate state with a partial rule set or no rules loaded. One consequence is that traffic that matches states that are valid in both the old and new rule set will not be disrupted.

Unless you've actually followed the advice from some of those old guides and *flushed* your existing rules (that *is* possible, using `pfctl -F all` or similar) before attempting to load a new one from your configuration file, the last valid configuration will remain loaded. In fact, flushing the rule set is rarely a good idea because it effectively puts your packet filter in a `pass all` mode, which in turn both opens the door to any comers and runs the risk of disrupting useful traffic while you're getting ready to load your rules.

Testing the Changed Rule Set

Once you have a rule set that `pfctl` loads without any errors, it's time to see whether the rules you've written behave as expected. Testing name resolution with a command such as `$ host nostarch.com`, as we did earlier, should still work. However, it's better to choose a domain you haven't accessed recently, such as one for a political party you wouldn't consider voting for, to be sure that you're not pulling DNS information from the cache.

You should be able to surf the Web and use several mail-related services, but due to the nature of this updated rule set, attempts to access TCP services other than the ones defined—SSH, SMTP, and so forth—on any remote system should fail. And, as with our simple rule set, your system should refuse all connections that don't match existing state table entries; only return traffic for connections initiated by this machine will be allowed in.

Displaying Information About Your System

The tests you've performed on your rule sets should have shown that PF is running and that your rules are behaving as expected. There are several ways to keep track of what happens in your running system. One of the more straightforward ways of extracting information about PF is to use the already familiar `pfctl` program.

Once PF is enabled and running, the system updates various counters and statistics in response to network activity. To confirm that PF is running and to view statistics about its activity, you can use `pfctl -s`, followed by the type of information you want to display. A long list of information types is available (see `man 8 pfctl` and look for the `-s` options). We'll get back to some of those display options in Chapter 9 and go into further detail about some of the statistics they provide in Chapter 10, when we use the data to optimize the configuration we're building.

The following shows an example of just the top part of the output of `pfctl -s info` (taken from my home gateway). The high-level information that indicates the system actually passes traffic can be found in this upper part.

```
$ sudo pfctl -s info
Status: Enabled for 24 days 12:11:27           Debug: err

Interface Stats for nfe0               IPv4             IPv6
  Bytes In                      43846385394                0
  Bytes Out                     20023639992               64
  Packets In
    Passed                         49380289                0
    Blocked                           49530                0
  Packets Out
    Passed                         45701100                1
    Blocked                            1034                0

State Table                           Total             Rate
  current entries                       319
  searches                        178598618           84.3/s
  inserts                           4965347            2.3/s
  removals                          4965028            2.3/s
```

The first line of the pfctl output indicates that PF is enabled and has been running for a little more than three weeks, which is equal to the time since I last performed a system upgrade that required a reboot.

The Interface Stats part of the display is an indication that the system's administrator has chosen one interface (here, nfe0) as the log interface for the system and shows the bytes in and out handled by the interface. If no log interface has been chosen, the display is slightly different. Now would be a good time to check what output your system produces. The next few items are likely to be more interesting in our context, showing the number of packets blocked or passed in each direction. This is where we find an early indication of whether the filtering rules we wrote are catching any traffic. In this case, either the rule set matches expected traffic well, or we have fairly well-behaved users and guests, with the number of packets passed being overwhelmingly larger than the number of packets blocked in both directions.

The next important indicator of a working system that's processing traffic is the block of State Table statistics. The state table current entries line shows that there are 319 active states or connections, while the state table has been searched (searches) for matches to existing states on average a little more than 84 times per second, for a total of just over 178 million times since the counters were reset. The inserts and removals counters show the number of times states have been created and removed, respectively. As expected, the number of insertions and removals differs by the number of currently active states, and the rate counters show that for the time since the counters were last reset, the rate of states created and removed matches exactly up to the resolution of this display.

The information here is roughly in line with the statistics you should expect to see on a gateway for a small network configured for IPv4 only. There's no reason to be alarmed by the packet passed in the IPv6 column. OpenBSD comes with IPv6 built in. During network interface configuration, by default, the TCP/IP stack sends IPv6 neighbor solicitation requests for the link local address. In a normal IPv4-only configuration, only the first few packets actually pass, and by the time the PF rule set from */etc/pf.conf* is fully loaded, IPv6 packets are blocked by the block all default rule. (In this example, they don't show up in `nfe0`'s statistics because IPv6 is tunneled over a different interface.)

Looking Ahead

You should now have a machine that can communicate with other Internet-connected machines, using a very basic rule set that serves as a starting point for controlling your network traffic. As you progress through this book, you'll learn how to add rules that do various useful things. In Chapter 3, we'll extend the configuration to act as a gateway for a small network. Serving the needs of several computers has some consequences, and we'll look at how to let at least some ICMP and UDP traffic through—for your own troubleshooting needs if nothing else.

In Chapter 3, we'll also consider network services that have consequences for your security, like FTP. Using packet filtering intelligently to handle services that are demanding, security-wise, is a recurring theme in this book.

3

INTO THE REAL WORLD

The previous chapter demonstrated the configuration for basic packet filtering on a single machine. In this chapter, we'll build on that basic setup but move into more conventional territory: the packet-filtering *gateway*. Although most of the items in this chapter are potentially useful in a single-machine setup, our main focus is to set up a gateway that forwards a selection of network traffic and handles common network services for a basic local network.

A Simple Gateway

We'll start with building what you probably associate with the term *firewall*: a machine that acts as a gateway for at least one other machine. In addition to forwarding packets between its various networks, this machine's mission will be to improve the signal-to-noise ratio in the network traffic it handles. That's where our PF configuration comes in.

But before diving into the practical configuration details, we need to dip into some theory and flesh out some concepts. Bear with me; this will end up saving you some headaches I've seen on mailing lists, newsgroups, and Web forums all too often.

Keep It Simple: Avoid the Pitfalls of in, out, and on

In the single-machine setup, life is relatively simple. Traffic you create should either pass out to the rest of the world or be blocked by your filtering rules, and you get to decide what you want to let in from elsewhere.

When you set up a gateway, your perspective changes. You go from the "It's me versus the network out there" mindset to "I'm the one who decides what to pass to or from all the networks I'm connected to." The machine has several, or at least two, network interfaces, each connected to a separate network, and its primary function (or at least the one we're interested in here) is to forward network traffic between networks. Conceptually, the network would look something like Figure 3-1.

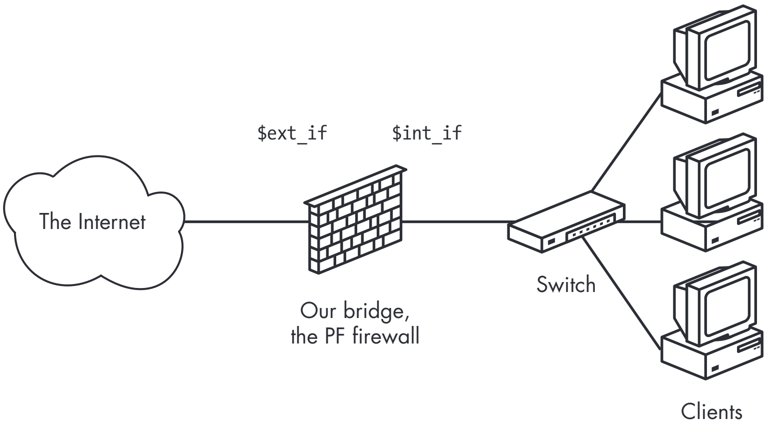

Figure 3-1: Network with a single gateway

It's very reasonable to think that if you want traffic to pass from the network connected to re1 to hosts on the network connected to re0, you'll need a rule like the following:[1]

```
pass in proto tcp on re1 from re1:network to re0:network \
                  port $ports keep state
```

However, one of the most common and most complained-about mistakes in firewall configuration is not realizing that the to keyword doesn't

1. In fact, the keep state part denotes the default behavior and is redundant if you're working with a PF version taken from OpenBSD 4.1 or later. However, there's generally no need to remove the specification from existing rules you come across when upgrading from earlier versions. To ease the transition, the examples in this book will make this distinction when needed.

in itself guarantee passage to the end point. The to keyword here means only that a packet or connection must have a destination address that matches those criteria in order to match the rule. The rule we just wrote lets the traffic pass in to just the gateway itself and on the specific interface named in the rule. To allow the packets in a bit further and to move on to the next network, we need a matching rule that says something like this:

```
pass out proto tcp on re0 from re1:network to re0:network \
                     port $ports keep state
```

But please stop and take a moment to read those rules one more time. This last rule allows only packets with a destination in the network directly connected to re0 to pass, and nothing else. If that's exactly what you want, fine. In other contexts, such rules are, while perfectly valid, more specific than the situation calls for. It's very easy to let yourself dive deeply into specific details and lose the higher-level view of the configuration's purpose—and maybe earn yourself a few extra rounds of debugging in the process.

If there are good reasons for writing very specific rules, like the preceding ones, you probably already know that you need them and why. By the time you have finished this book (if not a bit earlier), you should be able to articulate the circumstances when more specific rules are needed. However, for the basic gateway configurations in this chapter, it's likely that you'll want to write rules that are not interface-specific. In fact, in some cases, it isn't useful to specify the direction either; you'd simply use a rule like the following to let your local network access the Internet:

```
pass proto tcp from re1:network to port $ports keep state
```

For simple setups, interface-bound in and out rules are likely to add more clutter to your rule sets than they're worth. For a busy network admin, a readable rule set is a safer one. (And we'll look at some additional safety measures, like antispoof, in Chapter 10.)

For the remainder of this book, with some exceptions, we'll keep the rules as simple as possible for readability.

Network Address Translation vs. IPv6

Once we start handling traffic between separate networks, it's useful to look at how network addresses work and why you're likely to come across several different addressing schemes. The subject of network addresses has been a rich source of both confusion and buzzwords over the years. The underlying facts are sometimes hard to establish, unless you go to the source and wade through a series of RFCs. Over the next few paragraphs, I'll make an effort to clear up some of the confusion.

For example, a widely held belief is that if you have an internal network that uses a totally different address range from the one assigned to the interface attached to the Internet, you're safe, and no one from the outside can get at your network resources. This belief is closely related to the idea that the IP address of your firewall in the local network must be either `192.168.0.1` or `10.0.0.1`.

There's an element of truth in both notions, and those addresses are common defaults. But the real story is that it's possible to sniff one's way past network address translation, although PF offers some tricks that make that task harder.

The real reason we use a specific set of internal address ranges and a different set of addresses for unique external address ranges isn't primarily to address security concerns. Rather, it's the easiest way to work around a design problem in the Internet protocols: a limited range of possible addresses.

In the 1980s, when the Internet protocols were formulated, most computers on the Internet (or ARPANET, as it was known at the time) were large machines with anything from several dozen to several thousand users each. At the time, a 32-bit address space with more than four billion addresses seemed quite sufficient, but several factors have conspired to prove that assumption wrong. One factor is that the address-allocation process led to a situation where the largest chunks of the available address space were already allocated before some of the world's more populous nations even connected to the Internet. The other, and perhaps more significant, factor was that by the early 1990s, the Internet was no longer a research project, but rather a commercially available resource with consumers and companies of all sizes consuming IP address space at an alarming rate.

The long-term solution was to redefine the Internet to use a larger address space. In 1998, the specification for IPv6, with 128 bits of address space for a total of 2^{128} addresses, was published as RFC 2460. But while we were waiting for IPv6 to become generally available, we needed a stopgap solution. That solution came as a series of RFCs that specified how a gateway could forward traffic with IP addresses translated so that a large local network would look like just one computer to the rest of the Internet. Certain previously unallocated IP address ranges were set aside for these private networks. These were free for anyone to use, on the condition that traffic in those ranges wouldn't be allowed out on the Internet untranslated. Thus, *network address translation (NAT)* was born in the mid-1990s and quickly became the default way to handle addressing in local networks.[2]

PF supports IPv6 as well as the various IPv4 address translation tricks. (In fact, the BSDs were among the earliest IPv6 adopters, thanks to the efforts of the KAME project.[3]) All systems that have PF also support both the IPv4 and the IPv6 address families. If your IPv4 network needs a NAT

2. RFC 1631, "The IP Network Address Translator (NAT)," dated May 1994, and RFC 1918, "Address Allocation for Private Internets," dated February 1996, provide the details about NAT.

3. To quote the project home page at *http://www.kame.net/*, "The KAME project was a joint effort of six companies in Japan to provide a free stack of IPv6, IPsec, and Mobile IPv6 for BSD variants." The main research and development activities were considered complete in March 2006, with only maintenance activity continuing now that the important parts have been incorporated into the relevant systems.

configuration, you can integrate the translation as needed in your PF rule set. In other words, if you're using a system that supports PF, you can be reasonably sure that your IPv6 needs have been taken care of, at least on the operating-system level. However, some operating systems with a PF port use older versions of the code, and it's important to be aware that the general rule that newer PF code is better applies equally to the IPv6 context.

The examples in this book use mainly IPv4 addresses and NAT where appropriate, but most of the material is equally relevant to networks that have implemented IPv6.

Final Preparations: Defining Your Local Network

In Chapter 2, we set up a configuration for a single, standalone machine. We're about to extend that configuration to a gateway version, and it's useful to define a few more macros to help readability and to conceptually separate the local networks where you have a certain measure of control from everything else. So how do you define your "local" network in PF terms?

Earlier in this chapter, you saw the `*interface*:network` notation. This is a nice piece of shorthand, but you can make your rule set even more readable and easier to maintain by taking the macro a bit further. For example, you could define a `$localnet` macro as the network directly attached to your internal interface (`re1:network` in our examples). Or you could change the definition of `$localnet` to an IP address/netmask notation to denote a network, such as `192.168.100.0/24` for a subnet of private IPv4 addresses or `2001:db8:dead:beef::/64` for an IPv6 range.

If your network environment requires it, you could define your `$localnet` as a list of networks. For example, a sensible `$localnet` definition combined with `pass` rules that use the macro, such as the following, could end up saving you a few headaches:

```
pass proto { tcp, udp } from $localnet to port $ports
```

We'll stick to the convention of using macros such as `$localnet` for readability from here on.

Setting Up a Gateway

We'll take the single-machine configuration we built from the ground up in the previous chapter as our starting point for building our packet-filtering gateway. We assume that the machine has acquired another network card (or that you have set up a network connection from your local network to one or more other networks via Ethernet, PPP, or other means).

In our context, it isn't too interesting to look at the details of how the interfaces are configured. We just need to know that the interfaces are up and running.

For the following discussion and examples, only the interface names will differ between a PPP setup and an Ethernet one, and we'll do our best to get rid of the actual interface names as quickly as possible.

First, because packet forwarding is off by default in all BSDs, we need to turn it on in order to let the machine forward the network traffic it receives on one interface to other networks via one or more separate interfaces. Initially, we'll do this on the command line with a `sysctl` command for traditional IPv4:

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

If we need to forward IPv6 traffic, we use this `sysctl` command:

```
# sysctl net.inet6.ip6.forwarding=1
```

This is fine for now. However, in order for this to work once you reboot the computer at some time in the future, you need to enter these settings into the relevant configuration files.

In OpenBSD and NetBSD, you do this by editing */etc/sysctl.conf* and adding IP-forwarding lines to the end of the file so the last lines look like this:

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

In FreeBSD, make the change by putting these lines in your */etc/rc.conf*:

```
gateway_enable="YES" #for ipv4
ipv6_gateway_enable="YES" #for ipv6
```

The net effect is identical; the FreeBSD *rc* script sets the two values via `sysctl` commands. However, a larger part of the FreeBSD configuration is centralized into the *rc.conf* file.

Now it's time to check whether all of the interfaces you intend to use are up and running. Use **`ifconfig -a`** or **`ifconfig`** ***`interface_name`*** to find out.

The output of `ifconfig -a` on one of my systems looks like this:

```
$ ifconfig -a
lo0: flags=8049<UP,LOOPBACK,RUNNING,MULTICAST> mtu 33224
        groups: lo
        inet 127.0.0.1 netmask 0xff000000
        inet6 ::1 prefixlen 128
        inet6 fe80::1%lo0 prefixlen 64 scopeid 0x5
xl0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> mtu 1500
        lladdr 00:60:97:83:4a:01
        groups: egress
        media: Ethernet autoselect (100baseTX full-duplex)
        status: active
        inet 194.54.107.18 netmask 0xfffffff8 broadcast 194.54.107.23
        inet6 fe80::260:97ff:fe83:4a01%xl0 prefixlen 64 scopeid 0x1
```

```
fxp0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> mtu 1500
        lladdr 00:30:05:03:fc:41
        media: Ethernet autoselect (100baseTX full-duplex)
        status: active
        inet 194.54.103.65 netmask 0xffffffc0 broadcast 194.54.103.127
        inet6 fe80::230:5ff:fe03:fc41%fxp0 prefixlen 64 scopeid 0x2
pflog0: flags=141<UP,RUNNING,PROMISC> mtu 33224
enc0: flags=0<> mtu 1536
```

Your setup is most likely somewhat different. Here, the physical interfaces on the gateway are xl0 and fxp0. The logical interfaces lo0 (the loopback interface), enc0 (the encapsulation interface for IPSEC use), and pflog0 (the PF logging device) are probably on your system, too.

If you're on a dial-up connection, you probably use some variant of PPP for the Internet connection, and your external interface is the tun0 pseudo-device. If your connection is via some sort of broadband connection, you may have an Ethernet interface to play with. However, if you're in the significant subset of ADSL users who use PPP over Ethernet (PPPoE), the correct external interface will be one of the pseudo-devices tun0 or pppoe0 (depending on whether you use userland pppoe(8) or kernel mode pppoe(4)), not the physical Ethernet interface.

Depending on your specific setup, you may need to do some other device-specific configuration for your interfaces. After you have that set up, you can move on to the TCP/IP level and deal with the packet-filtering configuration.

If you still intend to allow any traffic initiated by machines on the inside, your */etc/pf.conf* for your initial gateway setup could look roughly like this:

```
ext_if = "re0" # macro for external interface - use tun0 or pppoe0 for PPPoE
int_if = "re1" # macro for internal interface
localnet = $int_if:network
# ext_if IPv4 address could be dynamic, hence ($ext_if)
match out on $ext_if inet from $localnet nat-to ($ext_if) # NAT, match IPv4 only
block all
pass from { self, $localnet }
```

Note the use of macros to assign logical names to the network interfaces. Here, Realtek Ethernet cards are used, but this is the last time we'll find this of any interest whatsoever in our context.

In truly simple setups like this one, we may not gain very much by using macros like these, but once the rule sets grow a little larger, you'll learn to appreciate the readability they add.

One possible refinement to this rule set would be to remove the macro ext_if and replace the $ext_if references with the string egress, which is the name of the interface group that contains the interface that has the default route. Interface groups are not macros, so you would write the name egress without a leading $ character.

Also note the match rule with nat-to. This is where you handle NAT from the nonroutable address inside your local network to the sole official address assigned to you. If your network uses official, routable IPv4 addresses, you simply leave this line out of your configuration. The match rules, which were introduced in OpenBSD 4.6, can be used to apply actions when a connection matches the criteria without deciding whether a connection should be blocked or passed.

The parentheses surrounding the last part of the match rule ($ext_if) are there to compensate for the possibility that the IP address of the external interface may be dynamically assigned. This detail will ensure that your network traffic runs without serious interruptions, even if the interface's IP address changes.

It's time to sum up the rule set we've built so far: (1) We block all traffic originating outside our own network. (2) We make sure all IPv4 traffic initiated by hosts in our local network will pass into the outside world only with the source address rewritten to the routable address assigned to the gateway's external interface. (3) Finally, we let all traffic from our local network (IPv4 and IPv6 both) and from the gateway itself pass. The keyword self in the final pass rule is a macro-ish reserved word in PF syntax that denotes all addresses assigned to all interfaces on the local host.

If your operating system runs a pre-OpenBSD 4.7 PF version, your first gateway rule set would look something like this:

```
ext_if = "re0" # macro for external interface - use tun0 or pppoe0 for PPPoE
int_if = "re1" # macro for internal interface
localnet = $int_if:network
# ext_if IP address could be dynamic, hence ($ext_if)
nat on $ext_if inet from $localnet to any -> ($ext_if)  # NAT, match IPv4 only
block all
pass from { self, $localnet } to any keep state
```

The nat rule here handles the translation much as does the match rule with nat-to in the previous example.

On the other hand, this rule set probably allows more traffic than you actually want to pass out of your network. In one of the networks where I've done a bit of work, the main part of the rule set is based on a macro called client_out:

```
client_out = "{ ftp-data, ftp, ssh, domain, pop3, auth, nntp, http,\
                https, 446, cvspserver, 2628, 5999, 8000, 8080 }"
```

It has this pass rule:

```
pass proto tcp from $localnet to port $client_out
```

This may be a somewhat peculiar selection of ports, but it's exactly what my colleagues there needed at the time. Some of the numbered ports were needed for systems that were set up for specific purposes at other sites. Your needs probably differ at least in some details, but this should cover some of the more useful services.

Here's another `pass` rule that is useful to those who want the ability to administer machines from elsewhere:

```
pass in proto tcp to port ssh
```

Or use this form, if you prefer:

```
pass in proto tcp to $ext_if port ssh
```

When you leave out the `from` part entirely, the default is `from any`, which is quite permissive. It lets you log in from anywhere, which is great if you travel a lot and need SSH access from unknown locations around the world. If you're not all that mobile—say you haven't quite developed the taste for conferences in far-flung locations, or you feel your colleagues can fend for themselves while you're on vacation—you may want to tighten up with a `from` part that includes only the places where you and other administrators are likely to log in from for legitimate reasons.

Our very basic rule set is still not complete. Next, we need to make the name service work for our clients. We start with another macro at the start of our rule set:

```
udp_services = "{ domain, ntp }"
```

This is supplemented with a rule that passes the traffic we want through our firewall:

```
pass quick proto { tcp, udp } to port $udp_services
```

Note the `quick` keyword in this rule. We've started writing rule sets that consist of several rules, and it's time to revisit the relationships and interactions between them.

As noted in the previous chapter, the rules are evaluated from top to bottom in the sequence they're written in the configuration file. For each packet or connection evaluated by PF, *the last matching rule* in the rule set is the one that's applied.

The `quick` keyword offers an escape from the ordinary sequence. When a packet matches a `quick` rule, the packet is treated according to the present rule. The rule processing stops without considering any further rules that might have matched the packet. As your rule sets grow longer and more complicated, you'll find this quite handy. For example, it's useful when you need a few isolated exceptions to your general rules.

This `quick` rule also takes care of NTP, which is used for time synchronization. Common to both the name service and time synchronization protocols is that they may, under certain circumstances, communicate alternately over TCP and UDP.

Testing Your Rule Set

You may not have gotten around to writing that formal test suite for your rule sets just yet, but there's every reason to test that your configuration works as expected.

The same basic tests in the standalone example from the previous chapter still apply. But now you need to test from the other hosts in your network as well as from your packet-filtering gateway. For each of the services you specified in your pass rules, test that machines in your local network get meaningful results. From any machine in your local network, enter a command like this:

```
$ host nostarch.com
```

It should return exactly the same results as when you tested the standalone rule set in the previous chapter, and traffic for the services you have specified should pass.[4]

You may not think it's necessary, but it doesn't hurt to check that the rule set works as expected from outside your gateway as well. If you've done exactly what this chapter says so far, it shouldn't be possible to contact machines in your local network from the outside.

WHY ONLY IP ADDRESSES— NOT HOSTNAMES OR DOMAIN NAMES?

Looking at the examples up to this point, you've probably noticed that the rule sets all have macros that expand into IP addresses or address ranges but never into hostnames or domain names. You're probably wondering why. After all, you've seen that PF lets you use service names in your rule set, so why not include hostnames or domain names?

The answer is that if you used domain names and hostnames in your rule set, the rule set would be valid only after the name service was running and accessible. In the default configuration, PF is loaded before any network services are running. This means that if you want to use domain names and hostnames in your PF configuration, you'll need to change the system's startup sequence (by editing */etc/rc.local*, perhaps) to load the name service–dependent rule set only after the name service is available. If you have only a limited number of hostnames or domain names you want to reference in your PF configuration, it's likely at least as useful to add those as IP addresses to name-mapping entries in your */etc/hosts* file and leave the *rc* scripts alone.

4. This is true unless, of course, the information changed in the meantime. Some sysadmins are fond of practical jokes, but most of the time changes in DNS zone information are due to real-world needs in that particular organization or network.

That Sad Old FTP Thing

The short list of real-life TCP ports we looked at a few moments back contained, among other things, *FTP*, the classic *file transfer protocol*. FTP is a relic of the early Internet, when experiments were the norm and security was not really on the horizon in any modern sense. FTP actually predates TCP/IP,[5] and it's possible to track the protocol's development through more than 50 RFCs. After more than 30 years, FTP is both a sad old thing and a problem child—emphatically so for anyone trying to combine FTP and firewalls. FTP is an old and weird protocol with a lot to dislike. Here are the most common points against it:

- Passwords are transferred in the clear.[6]
- The protocol demands the use of at least two TCP connections (control and data) on separate ports.
- When a session is established, data is communicated via ports usually selected at random.

All of these points make for challenges security-wise, even before considering any potential weaknesses in client or server software that may lead to security issues. As any network graybeard will tell you, these things tend to crop up when you need them the least.

Under any circumstances, other more modern and more secure options for file transfer exist, such as SFTP and SCP, which feature both authentication and data transfer via encrypted connections. Competent IT professionals should have a preference for some form of file transfer other than FTP.

Regardless of our professionalism and preferences, we sometimes must deal with things we would prefer not to use at all. In the case of FTP through firewalls, we can combat problems by redirecting the traffic to a small program that's written specifically for this purpose. The upside for us is that handling FTP offers us a chance to look at two fairly advanced PF features: *redirection* and *anchors*.

The easiest way to handle FTP in a default-to-block scenario such as ours is to have PF redirect the traffic for that service to an external application that acts as a *proxy* for the service. The proxy maintains its own named sub–rule set (an *anchor* in PF terminology), where it inserts or deletes rules as needed for the FTP traffic. The combination of redirection and the anchor provides a clean, well-defined interface between the packet-filtering subsystem and the proxy.

5. The earliest RFC describing the FTP protocol is RFC 114, dated April 10, 1971. The switch to TCP/IP happened with FTP version 5, as defined in RFCs 765 and 775, dated June and December 1980, respectively.

6. An encrypted version of the protocol, dubbed FTPS, is specified in RFC4217, but support remains somewhat spotty.

If We Must: ftp-proxy with Divert or Redirect

Enabling FTP transfers through your gateway is amazingly simple, thanks to the FTP-proxy program included in the OpenBSD base system. The program is called—you guessed it—`ftp-proxy`.

To enable `ftp-proxy`, you need to add this line to your */etc/rc.conf.local* file on OpenBSD:

```
ftpproxy_flags=""
```

On FreeBSD, */etc/rc.conf* needs to contain at least the first of these two lines:

```
ftpproxy_enable="YES"
ftpproxy_flags="" # and put any command line options here
```

If you need to specify any command-line options to `ftp-proxy`, you put them in the `ftpproxy_flags` variable.

You can start the proxy manually by running */usr/sbin/ftp-proxy* if you like (or even better, use the */etc/rc.d/ftp-proxy* script with the `start` option on OpenBSD), and you may want to do this in order to check that the changes to the PF configuration you're about to make have the intended effect.

For a basic configuration, you need to add only three elements to your */etc/pf.conf*: the anchor and two `pass` rules. The anchor declaration looks like this:

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

In pre-OpenBSD 4.7 versions, two anchor declarations were needed:

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

The proxy will insert the rules it generates for the FTP sessions here.

Then, you also need a `pass` rule to let FTP traffic into the proxy:

```
pass in quick inet proto tcp to port ftp divert-to 127.0.0.1 port 8021
```

Note the `divert-to` part. This redirects the traffic to the local port, where the proxy listens via the highly efficient, local-connections-only divert(4) interface. In OpenBSD versions 4.9 and older, the traffic diversion happened via an `rdr-to`. If you're upgrading an existing pre-OpenBSD 5.0 configuration, you'll need to update your `rdr-to` rules for the FTP proxy to use `divert-to` instead.

If your operating system uses a pre-OpenBSD 4.7 PF version, you need this version of the redirection rule:

```
rdr pass on $int_if inet proto tcp from any to any port ftp -> 127.0.0.1 port 8021
```

Finally, make sure your rule set contains a pass rule to let the packets pass from the proxy to the rest of the world, where $proxy expands to the address to which the proxy daemon is bound:

```
pass out inet proto tcp from $proxy to any port ftp
```

Reload your PF configuration:

```
$ sudo pfctl -f /etc/pf.conf
```

Before you know it, your users will thank you for making FTP work.

Variations on the ftp-proxy Setup

The preceding example covers a basic setup where the clients in your local network need to contact FTP servers elsewhere. This configuration should work well with most combinations of FTP clients and servers.

You can change the proxy's behavior in various ways by adding options to the ftpproxy_flags= line. You may bump into clients or servers with specific quirks that you need to compensate for in your configuration, or you may want to integrate the proxy in your setup in specific ways, such as assigning FTP traffic to a specific queue. For these and other finer points of ftp-proxy configuration, your best bet is to start by studying the man page.

If you're interested in ways to run an FTP server protected by PF and ftp-proxy, you could look into running a separate ftp-proxy in reverse mode (using the -R option) on a separate port with its own redirecting pass rule. It's even possible to set up the proxy to run in IPv6 mode, but if you're ahead of the pack in running the modern protocol, you're less likely to bother with FTP as your main file transfer protocol.

NOTE *If your PF version predates the ones described here, you're running on an outdated, unsupported operating system. I strongly urge you to schedule an operating system upgrade as soon as possible. If an upgrade is for some reason not an option, please look up the first edition of this book and study the documentation for your operating system for information on how to use some earlier FTP proxies.*

Making Your Network Troubleshooting-Friendly

Making your network troubleshooting-friendly is a potentially large subject. Generally, the debugging- or troubleshooting-friendliness of your TCP/IP network depends on how you treat the Internet protocol that was designed specifically with debugging in mind: ICMP.

ICMP is the protocol for sending and receiving *control messages* between hosts and gateways, mainly to provide feedback to a sender about any unusual or difficult conditions en route to the target host.

There's a lot of ICMP traffic, which usually happens in the background while you are surfing the Web, reading mail, or transferring files. Routers (remember, you're building one) use ICMP to negotiate packet sizes and other transmission parameters in a process often referred to as *path MTU discovery*.

You may have heard admins refer to ICMP as either "evil" or, if their understanding runs a little deeper, "a necessary evil." The reason for this attitude is purely historical. A few years back, it was discovered that the networking stacks of several operating systems contained code that could make the machine crash if it were sent a sufficiently large ICMP request.

One of the companies that was hit hard by this was Microsoft, and you can find a lot of material on the *ping of death* bug by using your favorite search engine. However, this all happened in the second half of the 1990s, and all modern operating systems have thoroughly sanitized their network code since then (at least, that's what we're led to believe).

One of the early work-arounds was to simply block either ICMP echo (ping) requests or even all ICMP traffic. That measure almost certainly led to poor performance and hard-to-debug network problems. In some places, however, these rule sets have been around for almost two decades, and the people who put them there are still scared. There's most likely little to no reason to worry about destructive ICMP traffic anymore, but here we'll look at how to manage just what ICMP traffic passes to or from your network.

In modern IPv6 networks, the updated `icmp6` protocol plays a more crucial role than ever in parameter passing and even host configuration, and network admins are playing a high-stakes game while learning the finer points of blocking or passing `icmp6` traffic. To a large extent, issues that are relevant for IPv4 ICMP generally apply to IPv6 ICMP6 as well, but in addition, ICMP6 is used for several mechanisms that were handled differently in IPv4. We'll dip into some of these issues after walking through the issues that are relevant for both IP protocol versions.

Do We Let It All Through?

The obvious question becomes, "If ICMP is such a good and useful thing, shouldn't we let it all through all the time?" The answer is that it depends.

Letting diagnostic traffic pass unconditionally makes debugging easier, of course, but it also makes it relatively easy for others to extract information about your network. So, a rule like the following might not be optimal if you want to cloak the internal workings of your IPv4 network:

```
pass inet proto icmp
```

If you want the same free flow of messages for your IPv6 traffic, the corresponding rule is this:

```
pass inet6 proto icmp6
```

In all fairness, it should also be said that you might find some ICMP and ICMP6 traffic quite harmlessly riding piggyback on your `keep state` rules.

The Easy Way Out: The Buck Stops Here

The easiest solution could very well be to allow all ICMP and ICMP6 traffic from your local network through and to let probes from elsewhere stop at your gateway:

```
pass inet proto icmp icmp-type $icmp_types from $localnet
pass inet6 proto icmp6 icmp6-type $icmp6_types from $localnet
pass inet proto icmp icmp-type $icmp_types to $ext_if
pass inet6 proto icmp6 icmp6-type $icmp6_types to $ext_if
```

This is assuming, of course that you've identified the list of desirable ICMP and ICMP6 types to fill out your macro definitions. We'll get back to those shortly. Stopping probes at the gateway might be an attractive option anyway, but let's look at a few other options that'll demonstrate some of PF's flexibility.

Letting ping Through

The rule set we have developed so far in this chapter has one clear disadvantage: Common troubleshooting commands, such as `ping` and `traceroute` (and their IPv6 equivalents, `ping6` and `traceroute6`), will not work. That may not matter too much to your users, and because it was the `ping` command that scared people into filtering or blocking ICMP traffic in the first place, there are apparently some people who feel we're better off without it. However, you'll find these troubleshooting tools useful. And with a couple of small additions to the rule set, they will be available to you.

The diagnostic commands `ping` and `ping6` rely on the ICMP and ICMP6 *echo request* (and the matching *echo reply*) types, and in order to keep our rule set tidy, we start by defining another set of macros:

```
icmp_types = "echoreq"
icmp6_types = "echoreq"
```

Then, we add rules that use the definitions:

```
pass inet proto icmp icmp-type $icmp_types
pass inet6 proto icmp6 icmp6-type $icmp6_types
```

The macros and the rules mean that ICMP and ICMP6 packets with type *echo request* will be allowed through and matching *echo replies* will be allowed to pass back due to PF's stateful nature. This is all the `ping` and `ping6` commands need in order to produce their expected results.

If you need more or other types of ICMP or ICMP6 packets to go through, you can expand `icmp_types` and `icmp6_types` to lists of those packet types you want to allow.

Helping traceroute

The `traceroute` command (and the IPv6 variant `traceroute6`) is useful when your users claim that the Internet isn't working. By default, Unix `traceroute` uses UDP connections according to a set formula based on destination. The following rules work with the `traceroute` and `traceroute6` commands on all forms of Unix I've had access to, including GNU/Linux:

```
# allow out the default range for traceroute(8):
# "base+nhops*nqueries-1" (33434+64*3-1)
pass out on egress inet proto udp to port 33433:33626 # For IPv4
pass out on egress inet6 proto udp to port 33433:33626 # For IPv6
```

This also gives you a first taste of what port ranges look like. They're quite useful in some contexts.

Experience so far indicates that `traceroute` and `traceroute6` implementations on other operating systems work roughly the same way. One notable exception is Microsoft Windows. On that platform, the *tracert.exe* program and its IPv6 sister *tracert6.exe* use ICMP echo requests for this purpose. So if you want to let Windows traceroutes through, you need only the first rule, much as when letting `ping` through. The Unix `traceroute` program can be instructed to use other protocols as well and will behave remarkably like its Microsoft counterpart if you use its `-I` command-line option. You can check the `traceroute` man page (or its source code, for that matter) for all the details.

This solution is based on a sample rule I found in an `openbsd-misc` post. I've found that list, and the searchable list archives (accessible among other places from *http://marc.info/*), to be a valuable resource whenever you need OpenBSD or PF-related information.

Path MTU Discovery

The last bit I'll remind you about when it comes to troubleshooting is the path MTU discovery. Internet protocols are designed to be device-independent, and one consequence of device independence is that you cannot always predict reliably what the optimal packet size is for a given connection. The main constraint on your packet size is called the *maximum transmission unit*, or *MTU*, which sets the upper limit on the packet size for an interface. The `ifconfig` command will show you the MTU for your network interfaces.

Modern TCP/IP implementations expect to be able to determine the correct packet size for a connection through a process that simply involves sending packets of varying sizes within the MTU of the local link with the "do not fragment" flag set. If a packet then exceeds the MTU somewhere along the way to the destination, the host with the lower MTU will return an ICMP packet indicating "type 3, code 4" and quoting its local MTU when the local upper limit has been reached. Now, you don't need to dive for the RFCs right away. Type 3 means *destination unreachable*, and code 4 is short for *fragmentation needed, but the "do not fragment" flag is set*. So if your connections to other networks, which may have MTUs that differ from your own, seem suboptimal, you could try changing your list of ICMP types slightly to let the IPv4 destination-unreachable packets through:

```
icmp_types = "{ echoreq, unreach }"
```

As you can see, this means you do not need to change the `pass` rule itself:

```
pass inet proto icmp icmp-type $icmp_types
```

Now I'll let you in on a little secret: In almost all cases, these rules aren't necessary for purposes of path MTU discovery (but they don't hurt either). However, even though the default PF `keep state` behavior takes care of most of the ICMP traffic you'll need, PF does let you filter on all variations of ICMP types and codes. For IPv6, you'd probably want to let the more common ICMP6 diagnostics through, such as the following:

```
icmp6_types = "{ echoreq unreach timex paramprob }"
```

This means that we let echo requests and destination unreachable, time exceeded, and parameter problem messages pass for IPv6 traffic. Thanks to the macro definitions, you don't need to touch the `pass` rule for the ICMP6 case either:

```
pass inet6 proto icmp6 icmp6-type $icmp6_types
```

But it's worth keeping in mind that IPv6 hosts rely on ICMP6 messages for automatic configuration-related tasks, and you may want to explicitly filter in order to allow or deny specific ICMP6 types at various points in your network.

For example, you'll want to let a router and its clients exchange router solicitation and router advertisement messages (ICMP6 type `routeradv` and `routersol`, respectively), while you may want to make sure that neighbor advertisements and neighbor solicitations (ICMP6 type `neighbradv` and `neighbrsol`, respectively) stay confined within their directly connected networks.

If you want to delve into more detail, the list of possible types and codes are documented in the `icmp(4)` and `icmp6(4)` man pages. The background information is available in the RFCs.[7]

7. The main RFCs describing ICMP and some related techniques are 792, 950, 1191, 1256, 2521, and 6145. ICMP updates for IPv6 are in RFC 3542 and RFC 4443. These documents are available in a number of places on the Web, such as *http://www.ietf.org/* and *http://www.faqs.org/*, and probably also via your package system.

Tables Make Your Life Easier

By now, you may be thinking that this setup gets awfully static and rigid. There will, after all, be some kinds of data relevant to filtering and redirection at a given time, but they don't deserve to be put into a configuration file! Quite right, and PF offers mechanisms for handling those situations.

Tables are one such feature. They're useful as lists of IP addresses that can be manipulated without reloading the entire rule set and also when fast lookups are desirable.

Table names are always enclosed in `< >`, like this:

```
table <clients> persist { 192.168.2.0/24, !192.168.2.5 }
```

Here, the network `192.168.2.0/24` is part of the table with one exception: The address `192.168.2.5` is excluded using the `!` operator (logical NOT). The keyword `persist` makes sure the table itself will exist, even if no rules currently refer to it.

It's also possible to load tables from files where each item is on a separate line, such as the file */etc/clients*:

```
192.168.2.0/24
!192.168.2.5
```

This, in turn, is used to initialize the table in */etc/pf.conf*:

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

So, for example, you can change one of our earlier rules to read like this to manage outgoing traffic from your client computers:

```
pass inet proto tcp from <clients> to any port $client_out
```

With this in hand, you can manipulate the table's contents live, like this:

```
$ sudo pfctl -t clients -T add 192.168.1/16
```

Note that this changes the in-memory copy of the table only, meaning that the change will not survive a power failure or reboot, unless you arrange to store your changes.

You might opt to maintain the on-disk copy of the table with a `cron` job that dumps the table content to disk at regular intervals, using a command such as the following:

```
$ sudo pfctl -t clients -T show >/etc/clients
```

Alternatively, you could edit the */etc/clients* file and replace the in-memory table contents with the file data:

```
$ sudo pfctl -t clients -T replace -f /etc/clients
```

For operations you'll be performing frequently, sooner or later, you'll end up writing shell scripts. It's likely that routine operations on tables, such as inserting or removing items or replacing table contents, will be part of your housekeeping scripts in the near future.

One common example is to enforce network access restrictions via `cron` jobs that replace the contents of the tables referenced as `from` addresses in the `pass` rules at specific times. In some networks, you may even need different access rules for different days of the week. The only real limitations lie in your own needs and your creativity.

We'll be returning to some handy uses of tables frequently over the next chapters, and we'll look at a few programs that interact with tables in useful ways.

4

WIRELESS NETWORKS MADE EASY

It's rather tempting to say that on BSD—and OpenBSD, in particular—there's no need to "make wireless networking easy" because it already is. Getting a wireless network running isn't very different from getting a wired one up and running, but there are some issues that turn up simply because we're dealing with radio waves and not wires. We'll look briefly at some of the issues before moving on to the practical steps involved in creating a usable setup.

Once we have covered the basics of getting a wireless network up and running, we'll turn to some of the options for making your wireless network more interesting and harder to break.

A Little IEEE 802.11 Background

Setting up any network interface, in principle, is a two-step process: You establish a link, and then you move on to configuring the interface for TCP/IP traffic.

In the case of wired Ethernet-type interfaces, establishing the link usually consists of plugging in a cable and seeing the link indicator light up. However, some interfaces require extra steps. Networking over dial-up connections, for example, requires telephony steps, such as dialing a number to get a carrier signal.

In the case of IEEE 802.11–style wireless networks, getting the carrier signal involves quite a few steps at the lowest level. First, you need to select the proper channel in the assigned frequency spectrum. Once you find a signal, you need to set a few link-level network identification parameters. Finally, if the station you want to link to uses some form of link-level encryption, you need to set the correct kind and probably negotiate some additional parameters.

Fortunately, on OpenBSD systems, all configuration of wireless network devices happens via `ifconfig` commands and options, as with any other network interface. While most network configuration happens via `ifconfig` on other BSDs, too, on some systems, specific features require other configuration.[1] Still, because we're introducing wireless networks here, we need to look at security at various levels in the networking stack from this new perspective.

There are basically three kinds of popular and simple IEEE 802.11 privacy mechanisms, and we'll discuss them briefly over the next sections.

NOTE *For a more complete overview of issues surrounding security in wireless networks, see Professor Kjell Jørgen Hole's articles and slides at* http://www.kjhole.com/ *and* http://www.nowires.org/.

MAC Address Filtering

The short version of the story about PF and MAC address filtering is that we don't do it. A number of consumer-grade, off-the-shelf wireless access points offer MAC address filtering, but contrary to common belief, they don't really add much security. The marketing succeeds largely because most consumers are unaware that it's possible to change the MAC address of essentially any wireless network adapter on the market today.[2]

1. On some systems, the older, device-specific programs, such as `wicontrol` and `ancontrol`, are still around, but for the most part, they are deprecated and have long been replaced with `ifconfig` functionality. On OpenBSD, the consolidation into `ifconfig` has been completed.

2. A quick man page lookup on OpenBSD will tell you that the command to change the MAC address for the interface `rum0` is simply `ifconfig rum0 lladdr 00:ba:ad:f0:0d:11`.

NOTE *If you really want to try MAC address filtering, you could look into using the* `bridge(4)` *facility and the bridge-related rule options in* `ifconfig(8)` *on OpenBSD 4.7 and later. We'll look at bridges and some of the more useful ways to use them with packet filtering in Chapter 5. Note that you can use the bridge filtering without really running a bridge by just adding one interface to the bridge.*

WEP

One consequence of using radio waves instead of wires to move data is that it's comparatively easier for outsiders to capture data in transit over radio waves. The designers of the 802.11 family of wireless network standards seem to have been aware of this fact, and they came up with a solution that they went on to market under the name *Wired Equivalent Privacy*, or *WEP*.

Unfortunately, the WEP designers came up with their wired equivalent encryption without actually reading up on recent research or consulting active researchers in the field. So the link-level encryption scheme they recommended is considered a pretty primitive homebrew among cryptography professionals. It was no great surprise when WEP encryption was reverse-engineered and cracked within a few months after the first products were released.

Even though you can download free tools to descramble WEP-encoded traffic in a matter of minutes, for a variety of reasons, WEP is still widely supported and used. Essentially, all IEEE 802.11 equipment available today has support for at least WEP, and a surprising number offer MAC address filtering, too.

You should consider network traffic protected only by WEP to be just marginally more secure than data broadcast in the clear. Then again, the token effort needed to crack into a WEP network may be sufficient to deter lazy and unsophisticated attackers.

WPA

It dawned on the 802.11 designers fairly quickly that their WEP system wasn't quite what it was cracked up to be, so they came up with a revised and slightly more comprehensive solution called *Wi-Fi Protected Access*, or *WPA*.

WPA looks better than WEP, at least on paper, but the specification is complicated enough that its widespread implementation was delayed. In addition, WPA has attracted its share of criticism over design issues and bugs that have produced occasional interoperability problems. Combined with the familiar issues of access to documentation and hardware, free software support varies. Most free systems have WPA support, and even though you may find that it's not available for all devices, the situation has been improving over time. If your project specification includes WPA, look carefully at your operating system and driver documentation.

And, of course, it goes almost without saying that you'll need further security measures, such as SSH or SSL encryption, to maintain any significant level of confidentiality for your data stream.

The Right Hardware for the Task

Picking the right hardware is not necessarily a daunting task. On a BSD system, the following simple command is all you need to enter to see a listing of all manual pages with the word *wireless* in their subject lines.[3]

```
$ apropos wireless
```

Even on a freshly installed system, this command will give you a complete list of all wireless network drivers available in the operating system.

The next step is to read the driver manual pages and compare the lists of compatible devices with what is available as parts or built into the systems you're considering. Take some time to think through your specific requirements. For test purposes, low-end `rum` or `ural` USB dongles (or the newer `urtwn` and `run`) will work and are quite convenient. Later, when you're about to build a more permanent infrastructure, you may want to look into higher-end gear, although you may find that the inexpensive test gear will perform quite well. Some wireless chipsets require firmware that for legal reasons can't be distributed on the OpenBSD install media. In most cases, the *fw_update* script will be able to fetch the required firmware on first boot after a successful install, as long as a network connection is available. If you install the units in an already configured system, you can try running *fw_update* manually. You may also want to read Appendix B of this book for some further discussion.

Setting Up a Simple Wireless Network

For our first wireless network, it makes sense to use the basic gateway configuration from the previous chapter as our starting point. In your network design, it's likely that the wireless network isn't directly attached to the Internet at large but that the wireless network will require a gateway of some sort. For that reason, it makes sense to reuse the working gateway setup for this wireless access point, with some minor modifications introduced over the next few paragraphs. After all, doing so is more convenient than starting a new configuration from scratch.

NOTE *We're in infrastructure-building mode here, and we'll be setting up the access point first. If you prefer to look at the client setup first, see "The Client Side" on page 55.*

The first step is to make sure you have a supported card and to check that the driver loads and initializes the card properly. The boot-time system messages scroll by on the console, but they're also recorded in the file

3. In addition, it's possible to look up man pages on the Web. Check *http://www.openbsd.org/* and the other project websites. They offer keyword-based man page searching.

/var/run/dmesg.boot. You can view the file itself or use the dmesg command to see these messages. With a successfully configured PCI card, you should see something like this:

```
ral0 at pci1 dev 10 function 0 "Ralink RT2561S" rev 0x00: apic 2 int 11 (irq
11), address 00:25:9c:72:cf:60
ral0: MAC/BBP RT2561C, RF RT2527
```

If the interface you want to configure is a hot-pluggable type, such as a USB or PC Card device, you can see the kernel messages by viewing the */var/log/messages* file—for example, by running tail -f on the file before you plug in the device.

Next, you need to configure the interface: first to enable the link and, finally, to configure the system for TCP/IP. You can do this from the command line, like this:

```
$ sudo ifconfig ral0 up mediaopt hostap mode 11g chan 1 nwid unwiredbsd nwkey 0x1deadbeef9
```

This command does several things at once. It configures the ral0 interface, enables the interface with the up parameter, and specifies that the interface is an access point for a wireless network with mediaopt hostap. Then, it explicitly sets the operating mode to 11g and the channel to 11. Finally, it uses the nwid parameter to set the network name to unwiredbsd, with the WEP key (nwkey) set to the hexadecimal string 0x1deadbeef9.

Use ifconfig to check that the command successfully configured the interface:

```
$ ifconfig ral0
ral0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> mtu 1500
        lladdr 00:25:9c:72:cf:60
        priority: 4
        groups: wlan
        media: IEEE802.11 autoselect mode 11g hostap
        status: active
        ieee80211: nwid unwiredbsd chan 1 bssid 00:25:9c:72:cf:60 nwkey <not displayed> 100dBm
        inet6 fe80::225:9cff:fe72:cf60%ral0 prefixlen 64 scopeid 0x2
```

Note the contents of the media and ieee80211 lines. The information displayed here should match what you entered on the ifconfig command line.

With the link part of your wireless network operational, you can assign an IP address to the interface. First, set an IPv4 address:

```
$ sudo ifconfig ral0 10.50.90.1 255.255.255.0
```

Setting an IPv6 is equally straightforward:

```
$ sudo ifconfig alias ral0 2001:db8::baad:f00d:1 64
```

On OpenBSD, you can combine both steps into one by creating a */etc/hostname.ral0* file, roughly like this:

```
up mediaopt hostap mode 11g chan 1 nwid unwiredbsd nwkey 0x1deadbeef9
inet6 alias 2001:db8::baad:f00d:1 64
```

Then, run **`sh /etc/netstart ral0`** (as root) or wait patiently for your next boot to complete.

Notice that the preceding configuration is divided over several lines. The first line generates an `ifconfig` command that sets up the interface with the correct parameters for the physical wireless network. The second line generates the command that sets the IPv4 address after the first command completes, followed by setting an IPv6 address for a dual-stack configuration. Because this is our access point, we set the channel explicitly, and we enable weak WEP encryption by setting the `nwkey` parameter.

On NetBSD, you can normally combine all of these parameters in one *rc.conf* setting:

```
ifconfig_ral0="mediaopt hostap mode 11g chan 1 nwid unwiredbsd nwkey
0x1deadbeef inet 10.50.90.1 netmask 255.255.255.0 inet6 2001:db8::baad:f00d:1
prefixlen 64 alias"
```

FreeBSD 8 and newer versions take a slightly different approach, tying wireless network devices to the unified `wlan(4)` driver. Depending on your kernel configuration, you may need to add the relevant module load lines to */boot/loader.conf*. On one of my test systems, */boot/loader.conf* looks like this:

```
if_rum_load="YES"
wlan_scan_ap_load="YES"
wlan_scan_sta_load="YES"
wlan_wep_load="YES"
wlan_ccmp_load="YES"
wlan_tkip_load="YES"
```

With the relevant modules loaded, setup is a multicommand affair, best handled by a *start_if.if* file for your wireless network. Here is an example of an */etc/start_if.rum0* file for a WEP access point on FreeBSD 8:

```
wlans_rum0="wlan0"
create_args_wlan0="wlandev rum0 wlanmode hostap"
ifconfig_wlan0="inet 10.50.90.1 netmask 255.255.255.0 ssid unwiredbsd \
wepmode on wepkey 0x1deadbeef9 mode 11g"
ifconfig_wlan0_ipv6="2001:db8::baad:f00d:1 prefixlen 64"
```

After a successful configuration, your `ifconfig` output should show both the physical interface and the `wlan` interface up and running:

```
rum0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> metric 0 mtu 2290
        ether 00:24:1d:9a:bf:67
        media: IEEE 802.11 Wireless Ethernet autoselect mode 11g <hostap>
        status: running
wlan0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> metric 0 mtu 1500
        ether 00:24:1d:9a:bf:67
        inet 10.50.90.1 netmask 0xffffff00 broadcast 10.50.90.255
        inet6 2001:db8::baad:f00d:1 prefixlen 64
        media: IEEE 802.11 Wireless Ethernet autoselect mode 11g <hostap>
        status: running
        ssid unwiredbsd channel 6 (2437 Mhz 11g) bssid 00:24:1d:9a:bf:67
        country US authmode OPEN privacy ON deftxkey UNDEF wepkey 1:40-bit
        txpower 0 scanvalid 60 protmode CTS dtimperiod 1 -dfs
```

The line `status: running` means that you're up and running, at least on the link level.

NOTE *Be sure to check the most up-to-date `ifconfig` man page for other options that may be more appropriate for your configuration.*

An OpenBSD WPA Access Point

WPA support was introduced in OpenBSD 4.4, with extensions to most wireless network drivers, and all basic WPA keying functionality was merged into `ifconfig(8)` in OpenBSD 4.9.

NOTE *There may still be wireless network drivers that don't have WPA support, so check the driver's man page to see whether WPA is supported before you try to configure your network to use it. You can combine 802.1*x *key management with an external authentication server for "enterprise" mode via the `security/wpa_supplicant` package, but we'll stick to the simpler preshared key setup for our purposes.*

The procedure for setting up an access point with WPA is quite similar to the one we followed for WEP. For a WPA setup with a preshared key (sometimes referred to as a *network password*), you would typically write a *hostname.if* file like this:

```
up media autoselect mediaopt hostap mode 11g chan 1 nwid unwiredbsd wpakey 0x1deadbeef9
inet6 alias 2001:db8::baad:f00d:1 64
```

If you're already running the WEP setup described earlier, disable those settings with the following:

```
$ sudo ifconfig ral0 -nwid -nwkey
```

Then, enable the new settings with this command:

```
$ sudo sh /etc/netstart ral0
```

You can check that the access point is up and running with ifconfig:

```
$ ifconfig ral0
ral0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> mtu 1500
        lladdr 00:25:9c:72:cf:60
        priority: 4
        groups: wlan
        media: IEEE802.11 autoselect mode 11g hostap
        status: active
        ieee80211: nwid unwiredbsd chan 1 bssid 00:25:9c:72:cf:60 wpapsk <not displayed>
wpaprotos wpa1,wpa2 wpaakms psk wpaciphers tkip,ccmp wpagroupcipher tkip 100dBm
        inet6 fe80::225:9cff:fe72:cf60%ral0 prefixlen 64 scopeid 0x2
        inet6 2001:db8::baad:f00d:1 prefixlen 64
        inet 10.50.90.1 netmask 0xff000000 broadcast 10.255.255.255
```

Note the status: active indication and that the WPA options we didn't set explicitly are shown with their sensible default values.

A FreeBSD WPA Access Point

Moving from the WEP access point we configured earlier to a somewhat safer WPA setup is straightforward. WPA support on FreeBSD comes in the form of hostapd (a program that is somewhat similar to OpenBSD's hostapd but not identical). We start by editing the */etc/start_if.rum0* file to remove the authentication information. The edited file should look something like this:

```
wlans_rum0="wlan0"
create_args_wlan0="wlandev rum0 wlanmode hostap"
ifconfig_wlan0="inet 10.50.90.1 netmask 255.255.255.0 ssid unwiredbsd mode 11g"
ifconfig_wlan0_ipv6="2001:db8::baad:f00d:1 prefixlen 64"
```

Next, we add the enable line for hostapd in */etc/rc.conf*:

```
hostapd_enable="YES"
```

And finally, hostapd will need some configuration of its own, in */etc/hostapd.conf*:

```
interface=wlan0
debug=1
ctrl_interface=/var/run/hostapd
ctrl_interface_group=wheel
ssid=unwiredbsd
wpa=1
wpa_passphrase=0x1deadbeef9
wpa_key_mgmt=WPA-PSK
wpa_pairwise=CCMP TKIP
```

Here, the interface specification is rather self-explanatory, while the debug value is set to produce minimal messages. The range is 0 through 4, where 0 is no debug messages at all. You shouldn't need to change the ctrl_interface* settings unless you're developing hostapd. The first of the next five lines sets the network identifier. The subsequent lines enable WPA and set the passphrase. The final two lines specify accepted key-management algorithms and encryption schemes. (For the finer details and updates, see the `hostapd(8)` and `hostapd.conf(5)` man pages.)

After a successful configuration (running `sudo /etc/rc.d/hostapd forcestart` comes to mind), `ifconfig` should produce output about the two interfaces similar to this:

```
rum0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> metric 0 mtu 2290
        ether 00:24:1d:9a:bf:67
        media: IEEE 802.11 Wireless Ethernet autoselect mode 11g <hostap>
        status: running
wlan0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> metric 0 mtu 1500
        ether 00:24:1d:9a:bf:67
        inet 10.50.90.1 netmask 0xffffff00 broadcast 10.50.90.255
        inet6 2001:db8::baad:f00d:1 prefixlen 64
        media: IEEE 802.11 Wireless Ethernet autoselect mode 11g <hostap>
        status: running
        ssid unwiredbsd channel 6 (2437 Mhz 11g) bssid 00:24:1d:9a:bf:67
        country US authmode WPA privacy MIXED deftxkey 2 TKIP 2:128-bit
        txpower 0 scanvalid 60 protmode CTS dtimperiod 1 -dfs
```

The line `status: running` means that you're up and running, at least on the link level.

The Access Point's PF Rule Set

With the interfaces configured, it's time to start configuring the access point as a packet-filtering gateway. You can start by copying the basic gateway setup from Chapter 3. Enable gatewaying via the appropriate entries in the access point's *sysctl.conf* or *rc.conf* file and then copy across the *pf.conf* file. Depending on the parts of the previous chapter that were most useful to you, the *pf.conf* file may look something like this:

```
ext_if = "re0" # macro for external interface - use tun0 or pppoe0 for PPPoE
int_if = "re1" # macro for internal interface
localnet = $int_if:network
# nat_address = 203.0.113.5 # Set addess for nat-to
client_out = "{ ssh, domain, pop3, auth, nntp, http,\
                https, cvspserver, 2628, 5999, 8000, 8080 }"
udp_services = "{ domain, ntp }"
icmp_types = "{ echoreq, unreach }"
# if IPv6, some ICMP6 accommodation is needed
icmp6_types = "{ echoreq unreach timex paramprob }"
# If ext_if IPv4 address is dynamic, ($ext_if) otherwise nat to specific address, ie
# match out on $ext_if inet from $localnet nat-to $nat_address
match out on $ext_if inet from $localnet nat-to ($ext_if)
block all
pass quick inet proto { tcp, udp } from $localnet to port $udp_services
pass log inet proto icmp icmp-type $icmp_types
pass inet6 proto icmp6 icmp6-type $icmp6_types
pass inet proto tcp from $localnet port $client_out
```

If you're running a PF version equal to OpenBSD 4.6 or earlier, the `match` rule with `nat-to` instead becomes this (assuming the external interface has one address, dynamically assigned):

```
nat on $ext_if from $localnet to any -> ($ext_if)
```

The only difference that's strictly necessary for your access point to work is the definition of `int_if`. You must change the definition of `int_if` to match the wireless interface. In our example, this means the line should now read as follows:

```
int_if = "ral0"  # macro for internal interface
```

More than likely, you'll also want to set up `dhcpd` to serve addresses and other relevant network information to IPv4 clients after they've associated with your access point. For IPv6 networks, you probably want to set up `rtadvd` (or even a DHCP6 daemon) to aid your IPv6 clients in their autoconfiguration. Setting up `dhcpd` and `rtadvd` is fairly straightforward if you read the man pages.

That's all there is to it. This configuration gives you a functional BSD access point, with at least token security (actually more like a *Keep Out!* sign) via WEP encryption or a slightly more robust link-level encryption with WPA. If you need to support FTP, copy the `ftp-proxy` configuration from the machine you set up in Chapter 3 and make changes similar to those you made for the rest of the rule set.

Access Points with Three or More Interfaces

If your network design dictates that your access point is also the gateway for a wired local network, or even several wireless networks, you need to make some minor changes to your rule set. Instead of just changing the value of the `int_if` macro, you might want to add another (descriptive) definition for the wireless interface, such as the following:

```
air_if = "ral0"
```

Your wireless interfaces are likely to be on separate subnets, so it might be useful to have a separate rule for each of them to handle any IPv4 NAT configuration. Here's an example for OpenBSD 4.7 and newer systems:

```
match out on $ext_if from $air_if:network nat-to ($ext_if)
```

And here's one on pre–OpenBSD 4.7 PF versions:

```
nat on $ext_if from $air_if:network to any -> ($ext_if) static-port
```

Depending on your policy, you might also want to adjust your `localnet` definition, or at least include `$air_if` in your `pass` rules where appropriate. And once again, if you need to support FTP, a separate pass with divert or redirection for the wireless network to `ftp-proxy` may be in order.

Handling IPSec, VPN Solutions

You can set up *virtual private networks (VPNs)* using built-in IPsec tools, OpenSSH, or other tools. However, due to the perceived poor security profile of wireless networks in general or for other reasons, you're likely to want to set up some additional security.

The options fall roughly into three categories:

SSH If your VPN is based on SSH tunnels, the baseline rule set already contains all the filtering you need. Your tunneled traffic will be indistinguishable from other SSH traffic to the packet filter.

IPsec with UDP key exchange (IKE/ISAKMP) Several IPsec variants depend critically on key exchange via `proto udp port 500` and use `proto udp port 4500` for *NAT Traversal (NAT-T)*. You need to let this traffic through in order to let the flows become established. Almost all implementations also depend critically on letting ESP protocol traffic (protocol number 50) pass between the hosts with the following:

```
pass proto esp from $source to $target
```

Filtering on IPsec encapsulation interfaces With a properly configured IPsec setup, you can set up PF to filter on the encapsulation interface `enc0` itself with the following:[4]

```
pass on enc0 proto ipencap from $source to $target keep state (if-bound)
```

See Appendix A for references to some of the more useful literature on the subject.

The Client Side

As long as you have BSD clients, setup is extremely easy. The steps involved in connecting a BSD machine to a wireless network are quite similar to the ones we just went through to set up a wireless access point. On OpenBSD, the configuration centers on the *hostname.if* file for the wireless interface. On FreeBSD, the configuration centers on *rc.conf* but will most likely involve a few other files, depending on your exact configuration.

4. In OpenBSD 4.8, the encapsulation interface became a cloneable interface, and you can configure several separate `enc` interfaces. All `enc` interfaces become members of the `enc` interface group.

OpenBSD Setup

Starting with the OpenBSD case, in order to connect to the WEP access point we just configured, your OpenBSD clients need a *hostname.if* (for example, */etc/hostname.ral0*) configuration file with these lines:

```
up media autoselect mode 11g chan 1 nwid unwiredbsd nwkey 0x1deadbeef9
dhcp
rtsol
```

The first line sets the link-level parameters in more detail than usually required. Only `up` and the `nwid` and `nwkey` parameters are strictly necessary. In almost all cases, the driver will associate with the access point on the appropriate channel and in the best available mode. The second line calls for a DHCP configuration and, in practice, causes the system to run a `dhclient` command to retrieve TCP/IP configuration information. The final line invokes `rtsol(8)` to initiate IPv6 configuration.

If you choose to go with the WPA configuration, the file will look like this instead:

```
up media autoselect mode 11g chan 1 nwid unwiredbsd wpakey 0x1deadbeef9
dhcp
rtsol
```

Again, the first line sets the link-level parameters, where the crucial ones are the network selection and encryption parameters `nwid` and `wpakey`. You can try omitting the `mode` and `chan` parameters; in almost all cases, the driver will associate with the access point on the appropriate channel and in the best available mode.

If you want to try out the configuration commands from the command line before committing the configuration to your */etc/hostname.if* file, the command to set up a client for the WEP network is as follows:

```
$ sudo ifconfig ral0 up mode 11g chan 1 nwid unwiredbsd nwkey 0x1deadbeef9
```

The `ifconfig` command should complete without any output. You can then use `ifconfig` to check that the interface was successfully configured. The output should look something like this:

```
$ ifconfig ral0
ral0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> mtu 1500
        lladdr 00:25:9c:72:cf:60
        priority: 4
        groups: wlan
        media: IEEE802.11 autoselect (OFDM54 mode 11g)
        status: active
        ieee80211: nwid unwiredbsd chan 1 bssid 00:25:9c:72:cf:60 nwkey <not displayed> 100dBm
        inet6 fe80::225:9cff:fe72:cf60%ral0 prefixlen 64 scopeid 0x2
```

Note that the ieee80211: line displays the network name and channel, along with a few other parameters. The information displayed here should match what you entered on the ifconfig command line.

Here is the command to configure your OpenBSD client to connect to the WPA network:

```
$ sudo ifconfig ral0 nwid unwiredbsd wpakey 0x1deadbeef9
```

The command should complete without any output. If you use ifconfig again to check the interface status, the output will look something like this:

```
$ ifconfig ral0
ral0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> mtu 1500
        lladdr 00:25:9c:72:cf:60
        priority: 4
        groups: wlan
        media: IEEE802.11 autoselect (OFDM54 mode 11g)
        status: active
        ieee80211: nwid unwiredbsd chan 1 bssid 00:25:9c:72:cf:60 wpapsk <not
displayed> wpaprotos wpa1,wpa2 wpaakms psk wpaciphers tkip,ccmp wpagroupcipher
tkip 100dBm
        inet6 fe80::225:9cff:fe72:cf60%ral0 prefixlen 64 scopeid 0x2
```

Check that the ieee80211: line displays the correct network name and sensible WPA parameters.

Once you are satisfied that the interface is configured at the link level, use the dhclient command to configure the interface for TCP/IP, like this:

```
$ sudo dhclient ral0
```

The dhclient command should print a summary of its dialogue with the DHCP server that looks something like this:

```
DHCPREQUEST on ral0 to 255.255.255.255 port 67
DHCPREQUEST on ral0 to 255.255.255.255 port 67
DHCPACK from 10.50.90.1 (00:25:9c:72:cf:60)
bound to 10.50.90.11 -- renewal in 1800 seconds.
```

To initialize the interface for IPv6, enter the following:

```
$ sudo rtsol ral0
```

The rtsol command normally completes without any messages. Check the interface configuration with ifconfig to see that the interface did in fact receive an IPv6 configuration.

FreeBSD Setup

On FreeBSD, you may need to do a bit more work than is necessary with OpenBSD. Depending on your kernel configuration, you may need to add the relevant module load lines to */boot/loader.conf*. On one of my test systems, */boot/loader.conf* looks like this:

```
if_rum_load="YES"
wlan_scan_ap_load="YES"
wlan_scan_sta_load="YES"
wlan_wep_load="YES"
wlan_ccmp_load="YES"
wlan_tkip_load="YES"
```

With the relevant modules loaded, you can join the WEP network we configured earlier by issuing the following command:

```
$ sudo ifconfig wlan create wlandev rum0 ssid unwiredbsd wepmode on wepkey 0x1deadbeef9 up
```

Then, issue this command to get an IPv4 configuration for the interface:

```
$ sudo dhclient wlan0
```

To initialize the interface for IPv6, enter the following:

```
$ sudo rtsol ral0
```

The `rtsol` command normally completes without any messages. Check the interface configuration with `ifconfig` to see that the interface did in fact receive an IPv6 configuration.

For a more permanent configuration, create a *start_if.rum0* file (replace *rum0* with the name of the physical interface if it differs) with content like this:

```
wlans_rum0="wlan0"
create_args_wlan0="wlandev rum0 ssid unwiredbsd wepmode on wepkey 0x1deadbeef9 up"
ifconfig_wlan0="DHCP"
ifconfig_wlan0_ipv6="inet6 accept_rtadv"
```

If you want to join the WPA network, you need to set up `wpa_supplicant` and change your network interface settings slightly. For the WPA access point, connect with the following configuration in your *start_if.rum0* file:

```
wlans_rum0="wlan0"
create_args_wlan0="wlandev rum0"
ifconfig_wlan0="WPA"
```

You also need an */etc/wpa_supplicant.conf* file that contains the following:

```
network={
  ssid="unwiredbsd"
  psk="0x1deadbeef9"
}
```

Finally, add a second `ifconfig_wlan0` line in *rc.conf* to ensure that `dhclient` runs correctly.

```
ifconfig_wlan0="DHCP"
```

For the IPv6 configuration, add the following line to *rc.conf*:

```
ifconfig_wlan0_ipv6="inet6 accept_rtadv"
```

Other WPA networks may require additional options. After a successful configuration, the `ifconfig` output should display something like this:

```
rum0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> metric 0 mtu 2290
        ether 00:24:1d:9a:bf:67
        media: IEEE 802.11 Wireless Ethernet autoselect mode 11g
        status: associated
wlan0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> metric 0 mtu 1500
        ether 00:24:1d:9a:bf:67
        inet 10.50.90.16 netmask 0xffffff00 broadcast 10.50.90.255
        inet6 2001:db8::baad:f00d:1635 prefixlen 64
        media: IEEE 802.11 Wireless Ethernet OFDM/36Mbps mode 11g
        status: associated
        ssid unwiredbsd channel 1 (2412 Mhz 11g) bssid 00:25:9c:72:cf:60
        country US authmode WPA2/802.11i privacy ON deftxkey UNDEF
        TKIP 2:128-bit txpower 0 bmiss 7 scanvalid 450 bgscan bgscanintvl 300
        bgscanidle 250 roam:rssi 7 roam:rate 5 protmode CTS roaming MANUAL
```

Guarding Your Wireless Network with authpf

Security professionals tend to agree that even though WEP encryption offers little protection, it's just barely enough to signal to would-be attackers that you don't intend to let all and sundry use your network resources. Using WPA increases security significantly, at the cost of some complexity in contexts that require the "enterprise"-grade options.

The configurations we've built so far in this chapter are functional. Both the WEP and WPA configurations will let all reasonably configured wireless clients connect, and that may be a problem in itself because that configuration doesn't have any real support built in for letting you decide who uses your network.

As mentioned earlier, MAC address filtering is not really a solid defense against attackers because changing a MAC address is just too easy. The OpenBSD developers chose a radically different approach to this problem when they introduced `authpf` in OpenBSD version 3.1. Instead of tying access to a hardware identifier, such as the network card's MAC address, they decided that the robust and highly flexible user authentication mechanisms already in place were more appropriate for the task. The user shell `authpf` lets the system load PF rules on a per-user basis, effectively deciding which user gets to do what.

To use `authpf`, you create users with the `authpf` program as their shell. In order to get network access, the user logs in to the gateway using SSH. Once the user successfully completes SSH authentication, `authpf` loads the rules you have defined for the user or the relevant class of users.

These rules, which usually are written to apply only to the IP address the user logged in from, stay loaded and in force for as long as the user stays logged in via the SSH connection. Once the SSH session is terminated, the rules are unloaded, and in most scenarios, all non-SSH traffic from the user's IP address is denied. With a reasonable setup, only traffic originated by authenticated users will be let through.

NOTE *On OpenBSD,* `authpf` *is one of the login classes offered by default, as you'll notice the next time you create a user with* `adduser`.

For systems where the `authpf` login class isn't available by default, you may need to add the following lines to your */etc/login.conf* file:

```
authpf:\
        :welcome=/etc/motd.authpf:\
        :shell=/usr/sbin/authpf:\
        :tc=default:
```

The next couple of sections contain a few examples that may or may not fit your situation directly but that I hope will give you ideas you can use.

A Basic Authenticating Gateway

Setting up an authenticating gateway with `authpf` involves creating and maintaining a few files besides your basic *pf.conf*. The main addition is *authpf.rules*. The other files are fairly static entities that you won't be spending much time on once they've been created.

Start by creating an empty */etc/authpf/authpf.conf* file. This file needs to be there in order for `authpf` to work, but it doesn't actually need any content, so creating an empty file with touch is appropriate.

The other relevant bits of */etc/pf.conf* follow. First, here are the interface macros:

```
ext_if = "re0"
int_if = "athn0"
```

In addition, if you define a table called `<authpf_users>`, `authpf` will add the IP addresses of authenticated users to the table:

```
table <authpf_users> persist
```

If you need to run NAT, the rules that take care of the translation could just as easily go in *authpf.rules*, but keeping them in the *pf.conf* file doesn't hurt in a simple setup like this:

```
pass out on $ext_if inet from $localnet nat-to ($ext_if)
```

Here's pre–OpenBSD 4.7 syntax:

```
nat on $ext_if inet from $localnet to any -> ($ext_if)
```

Next, we create the `authpf` anchor, where rules from *authpf.rules* are loaded once the user authenticates:

```
anchor "authpf/*"
```

For pre–OpenBSD 4.7 `authpf` versions, several anchors were required, so the corresponding section would be as follows:

```
nat-anchor "authpf/*"
rdr-anchor "authpf/*"
binat-anchor "authpf/*"
anchor "authpf/*"
```

This brings us to the end of the required parts of a *pf.conf* file for an `authpf` setup.

For the filtering part, we start with the block all default and then add the `pass` rules we need. The only essential item at this point is to let SSH traffic pass on the internal network:

```
pass quick on $int_if proto { tcp, udp } to $int_if port ssh
```

From here on out, it really is up to you. Do you want to let your clients have name resolution before they authenticate? If so, put the `pass` rules for the TCP and UDP service domain in your *pf.conf* file, too.

For a relatively simple and egalitarian setup, you could include the rest of our baseline rule set, changing the `pass` rules to allow traffic from the addresses in the `<authpf_users>` table, rather than any address in your local network:

```
pass quick proto { tcp, udp } from <authpf_users> to port $udp_services
pass proto tcp from <authpf_users> to port $client_out
```

For a more differentiated setup, you could put the rest of your rule set in */etc/authpf/authpf.rules* or per-user rules in customized *authpf.rules* files in each user's directory under */etc/authpf/users/*. If your users generally need some protection, your general */etc/authpf/authpf.rules* could have content like this:

```
client_out = "{ ssh, domain, pop3, auth, nntp, http, https }"
udp_services = "{ domain, ntp }"
pass quick proto { tcp, udp } from $user_ip to port $udp_services
pass proto tcp from $user_ip to port $client_out
```

The macro `user_ip` is built into `authpf` and expands to the IP address from which the user authenticated. These rules will apply to any user who completes authentication at your gateway.

A nice and relatively easy addition to implement is special-case rules for users with different requirements than your general user population. If an *authpf.rules* file exists in the user's directory under */etc/authpf/users/*, the rules in that file will be loaded for the user. This means that your naive user Peter who only needs to surf the Web and have access to a service that runs on a high port on a specific machine could get what he needs with a */etc/authpf/users/peter/authpf.rules* file like this:

```
client_out = "{ domain, http, https }"
pass inet from $user_ip to 192.168.103.84 port 9000
pass quick inet proto { tcp, udp } from $user_ip to port $client_out
```

On the other hand, Peter's colleague Christina runs OpenBSD and generally knows what she's doing, even if she sometimes generates traffic to and from odd ports. You could give her free rein by putting this in */etc/authpf/users/christina/authpf.rules*:

```
pass from $user_ip os = "OpenBSD" to any
```

This means Christina can do pretty much anything she likes over TCP/IP as long as she authenticates from her OpenBSD machines.

Wide Open but Actually Shut

In some settings, it makes sense to set up your network to be open and unencrypted at the link level, while enforcing some restrictions via `authpf`. The next example is very similar to Wi-Fi zones you may encounter in airports or other public spaces, where anyone can associate to the access points and get an IP address, but any attempt at accessing the Web will be redirected to one specific Web page until the user has cleared some sort of authentication.[5]

5. Thanks to Vegard Engen for the idea and for showing me his configuration, which is preserved here in spirit, if not in all its details.

This *pf.conf* file is again built on our baseline, with two important additions to the basic authpf setup—a macro and a redirection:

```
ext_if = "re0"
int_if = "ath0"
auth_web="192.168.27.20"
dhcp_services = "{ bootps, bootpc }" # DHCP server + client
table <authpf_users> persist
pass in quick on $int_if proto tcp from ! <authpf_users> to port http rdr-to $auth_web
match out on $ext_if from $int_if:network nat-to ($ext_if)
anchor "authpf/*"
block all
pass quick on $int_if inet proto { tcp, udp } to $int_if port $dhcp_services
pass quick inet proto { tcp, udp } from $int_if:network to any port domain
pass quick on $int_if inet proto { tcp, udp } to $int_if port ssh
For older authpf versions, use this file instead:
ext_if = "re0"
int_if = "ath0"
auth_web="192.168.27.20"
dhcp_services = "{ bootps, bootpc }" # DHCP server + client
table <authpf_users> persist
rdr pass on $int_if proto tcp from ! <authpf_users> to any port http -> $auth_web
nat on $ext_if from $localnet to any -> ($ext_if)
nat-anchor "authpf/*"
rdr-anchor "authpf/*"
binat-anchor "authpf/*"
anchor "authpf/*"
block all
pass quick on $int_if inet proto { tcp, udp } to $int_if port $dhcp_services
pass quick inet proto { tcp, udp } from $int_if:network to port domain
pass quick on $int_if inet proto { tcp, udp } to $int_if port ssh
```

The auth_web macro and the redirection make sure all Web traffic from addresses that are not in the <authpf_users> table leads all nonauthenticated users to a specific address. At that address, you set up a Web server that serves up whatever you need. This could range from a single page with instructions on whom to contact in order to get access to the network all the way up to a system that accepts credit cards and handles user creation.

Note that in this setup, name resolution will work, but all surfing attempts will end up at the auth_web address. Once the users clear authentication, you can add general rules or user-specific ones to the *authpf.rules* files as appropriate for your situation.

5

BIGGER OR TRICKIER NETWORKS

In this chapter, we'll build on the material in previous chapters to meet the real-life challenges of both large and small networks with relatively demanding applications or users. The sample configurations in this chapter are based on the assumption that your packet-filtering setups will need to accommodate services you run on your local network. We'll mainly look at this challenge from a Unix perspective, focusing on SSH, email, and Web services (with some pointers on how to take care of other services).

This chapter is about the things to do when you need to combine packet filtering with services that must be accessible outside your local network. How much this complicates your rule sets will depend on your network design and, to a certain extent, on the number of routable addresses you have available. We'll begin with configurations for official, routable IPv4 addresses as well as the generally roomier IPv6 address ranges. Then, we'll move on to situations with as few as one routable IPv4 address and the PF-based work-arounds that make the services usable even under these restrictions.

A Web Server and Mail Server on the Inside: Routable IPv4 Addresses

How complicated is your network? How complicated does it need to be?

We'll start with the baseline scenario of the sample clients from Chapter 3. We set up the clients behind a basic PF firewall and give them access to a range of services hosted elsewhere but no services running on the local network. These clients get three new neighbors: a mail server, a Web server, and a file server. In this scenario, we use official, routable IPv4 addresses because it makes life a little easier. Another advantage of this approach is that with routable addresses, we can let two of the new machines run DNS for our *example.com* domain: one as the master and the other as an authoritative slave.[1] And as you'll see, adding IPv6 addresses and running a dual-stack network won't necessarily make your rule set noticeably more complicated.

NOTE *For DNS, it always makes sense to have at least one authoritative slave server somewhere outside your own network (in fact, some top-level domains won't let you register a domain without it). You may also want to arrange for a backup mail server to be hosted elsewhere. Keep these things in mind as you build your network.*

At this stage, we keep the physical network layout fairly simple. We put the new servers in the same local network as the clients—possibly in a separate server room but certainly on the same network segment or switch as the clients. Conceptually, the new network looks something like Figure 5-1.

With the basic parameters for the network in place, we can start setting up a sensible rule set for handling the services we need. Once again, we start from the baseline rule set and add a few macros for readability.

The macros we need come rather naturally from the specifications:

- Web server:

```
webserver = "{ 192.0.2.227, 2001:db8::baad:f00d:f17 }"
```

- Web server services:

```
webports = "{ http, https }"
```

- Mail server:

```
emailserver = "{ 192.0.2.225, 2001:db8::baad:f00d:f117 }"
```

1. In fact, the *example.com* network here lives in the 192.0.2.0/24 block, which is reserved in RFC 3330 for example and documentation use. We use this address range mainly to differentiate from the NAT examples elsewhere in this book, which use addresses in the "private" RFC 1918 address space.

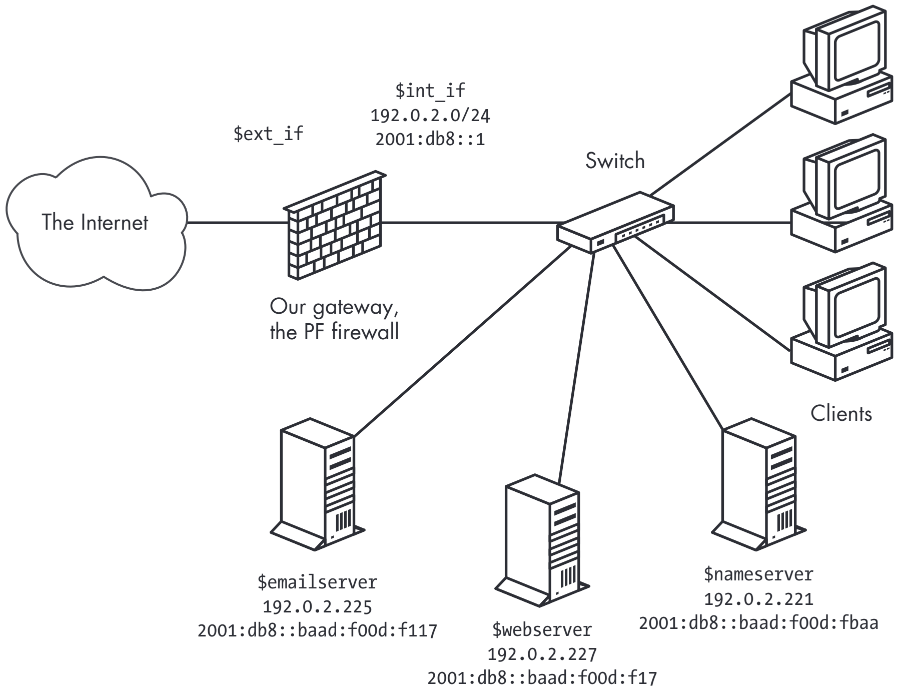

Figure 5-1: A basic network with servers and clients on the inside

- Mail server services:

```
email = "{ smtp, pop3, imap, imap3, imaps, pop3s }"
```

- Name servers:

```
nameservers = "{ 192.0.2.221,  192.0.2.223 , \
                 2001:db8::baad:f00d:fbaa, 2001:db8::baad:f00d:ff00 }"
```

NOTE *At this point, you've probably noticed that both the IPv4 and IPv6 addresses for our servers are placed fairly close together within their respective address ranges. Some schools of thought hold that in the case of IPv6, each interface should be allocated at least a /64 range if your total allocation can bear it. Others have advocated more modest allocations. The IETF's current best practice document on the matter is RFC6177, available from the IETF website (*http://www.ietf.org*).*

We assume that the file server doesn't need to be accessible to the outside world, unless we choose to set it up with a service that needs to be visible outside the local network, such as an authoritative slave name server for our domain. Then, with the macros in hand, we add the pass rules. Starting with the Web server, we make it accessible to the world with the following:

```
pass proto tcp to $webserver port $webports
```

IS SYNPROXY WORTH THE TROUBLE?

Over the years, the `synproxy state` option has received a lot of attention as a possible bulwark against ill-intentioned traffic from the outside. Specifically, the `synproxy state` option was intended to protect against SYN-flood attacks that could lead to resource exhaustion at the back end.

It works like this: When a new connection is created, PF normally lets the communication partners handle the connection setup themselves, simply passing the packets on if they match a `pass` rule. With `synproxy` enabled, PF handles the initial connection setup and hands over the connection to the communication partners only once it's properly established, essentially creating a buffer between the communication partners. The SYN proxying is slightly more expensive than the default `keep state`, but not necessarily noticeably so on reasonably scaled equipment.

The potential downsides become apparent in load-balancing setups where a SYN-proxying PF could accept connections that the backend isn't ready to accept, in some cases short-circuiting the redundancy by setting up connections to hosts other than those the load-balancing logic would have selected. The classic example here is a pool of HTTP servers with round-robin DNS. But the problem becomes especially apparent in protocols like SMTP, where the built-in redundancy dictates (by convention, at least—the actual RFC is a bit ambiguous) that if a primary mail exchanger isn't accepting connections, you should try a secondary instead.

When considering a setup where `synproxy` seems attractive, keep these issues in mind and analyze the potential impact on your setup that would come from adding `synproxy` to the mix. If you conclude that SYN proxying is needed, simply tack on `synproxy state` at the end of the rules that need the option. The rule of thumb is, if you are under active attack, inserting the `synproxy` option may be useful as a temporary measure. Under normal circumstances, it isn't needed as a permanent part of your configuration.

On a similar note, we let the world talk to the mail server:

```
pass proto tcp to $emailserver port $email
```

This lets clients anywhere have the same access as the ones in your local network, including a few mail-retrieval protocols that may run without encryption. That's common enough in the real world, but you might want to consider your options if you're setting up a new network.

For the mail server to be useful, it needs to be able to send mail to hosts outside the local network, too:

```
pass log proto tcp from $emailserver to port smtp
```

Keep in mind that the rule set starts with a `block all` rule, which means that only the mail server is allowed to initiate SMTP traffic from the local network to the rest of the world. If any of the other hosts on the network need to send email to or receive email from the outside world, they need to use the designated mail server. This could be a good way to ensure, for example, that you make it as hard as possible for any spam-sending zombie machines that might turn up in your network to deliver their payloads.

Finally, the name servers need to be accessible to clients outside our network who look up the information about *example.com* and any other domains for which we answer authoritatively:

```
pass proto { tcp, udp } to $nameservers port domain
```

Having integrated all the services that need to be accessible from the outside world, our rule set ends up looking roughly like this:

```
ext_if = "ep0" # macro for external interface - use tun0 or pppoe0 for PPPoE
int_if = "ep1"  # macro for internal interface
localnet = $int_if:network
webserver = "{ 192.0.2.227, 2001:db8::baad:f00d:f17 }"
webports = "{ http, https }"
emailserver = "{ 192.0.2.225, 2001:db8::baad:f00d:f117 }"
email = "{ smtp, pop3, imap, imap3, imaps, pop3s }"
nameservers = "{ 192.0.2.221, 192.0.2.223, \
                 2001:db8::baad:f00d:fbaa, 2001:db8::baad:f00d:ff00 }"
client_out = "{ ssh, domain, pop3, auth, nntp, http,\
                https, cvspserver, 2628, 5999, 8000, 8080 }"
udp_services = "{ domain, ntp }"
icmp_types = "{ echoreq, unreach }"
icmp6_types = "{ echoreq unreach timex paramprob }"
block all
pass quick proto { tcp, udp } from $localnet to port $udp_services
pass log inet proto icmp all icmp-type $icmp_types
pass inet6 proto icmp6 icmp6-type $icmp6_types
pass proto tcp from $localnet to port $client_out
pass proto { tcp, udp } to $nameservers port domain
pass proto tcp to $webserver port $webports
pass log proto tcp to $emailserver port $email
pass log proto tcp from $emailserver to port smtp
```

This is still a fairly simple setup, but unfortunately, it has one potentially troubling security disadvantage. The way this network is designed, the servers that offer services to the world at large are all *in the same local network* as your clients, and you'd need to restrict any internal services to only local access. In principle, this means that an attacker would need to compromise only one host in your local network to gain access to any resource there, putting the miscreant on equal footing with any user in your local network. Depending on how well each machine and resource are protected from unauthorized access, this could be anything from a minor annoyance to a major headache.

In the next section, we'll look at some options for segregating the services that need to interact with the world at large from the local network.

A Degree of Separation: Introducing the DMZ

In the previous section, you saw how to set up services on your local network and make them selectively available to the outside world through a sensible PF rule set. For more fine-grained control over access to your internal network, as well as the services you need to make it visible to the rest of the world, add a degree of physical separation. Even a separate *virtual local area network (VLAN)* will do nicely.

Achieving the physical and logical separation is fairly easy: Simply move the machines that run the public services to a separate network that's attached to a separate interface on the gateway. The net effect is a separate network that isn't quite part of your local network but isn't entirely in the public part of the Internet either. Conceptually, the segregated network looks like Figure 5-2.

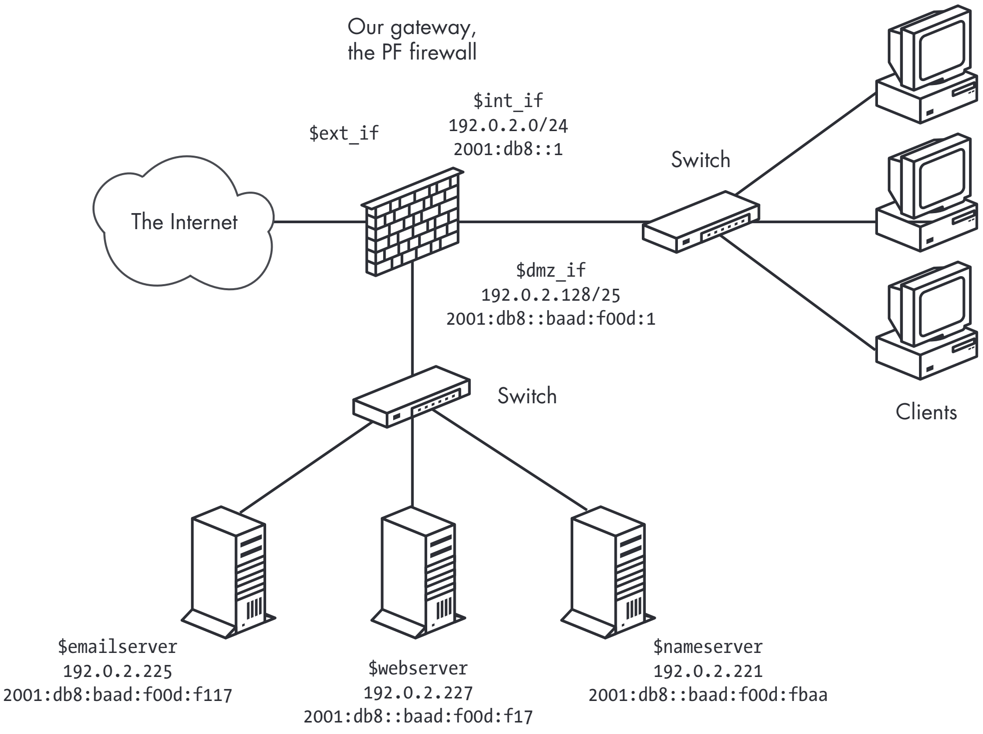

Figure 5-2: A network with the servers in a DMZ

NOTE *Think of this little network as a zone of relative calm between the territories of hostile factions. It's no great surprise that a few years back, someone coined the phrase* demilitarized zone (DMZ) *to describe this type of configuration.*

For address allocation, you can segment off an appropriately sized chunk of your official address space for the new DMZ network. Alternatively, you can move those parts of your network that don't have a specific need to run with publicly accessible and routable IPv4 addresses into a NAT environment. Either way, you end up with at least one more interface in your filtering configuration. As you'll see later, if you're really short of official IPv4 addresses, it's possible to run a DMZ setup in all-NAT environments as well.

The adjustments to the rule set itself don't need to be extensive. If necessary, you can change the configuration for each interface. The basic rule-set logic remains, but you may need to adjust the definitions of the macros (`webserver`, `mailserver`, `nameservers`, and possibly others) to reflect your new network layout.

In our example, we could choose to segment off the part of our address ranges where we've already placed our servers. If we leave some room for growth, we can set up the IPv4 range for the new `dmz_if` on a /25 subnet with a network address and netmask of 192.0.2.128/255.255.255.128. This leaves us with 192.0.2.129 through 192.0.2.254 as the usable address range for hosts in the DMZ. As we've already placed our servers in the 2001:db8::baad:f00d:0/112 network (with a measly 65,536 addresses to play with), the easiest way forward for the IPv6 range is to segment off that network, too, and assign the interface facing the network an appropriate IPv6 address, like the one in Figure 5-2.

With that configuration and no changes in the IP addresses assigned to the servers, you don't really need to touch the rule set at all for the packet filtering to work after setting up a physically segregated DMZ. That's a nice side effect, which could be due to either laziness or excellent long-range planning. Either way, it underlines the importance of having a sensible address-allocation policy in place.

It might be useful to tighten up your rule set by editing your `pass` rules so the traffic to and from your servers is allowed to pass only on the interfaces that are actually relevant to the services:

```
pass in on $ext_if proto { tcp, udp } to $nameservers port domain
pass in on $int_if proto { tcp, udp } from $localnet to $nameservers \
     port domain
pass out on $dmz_if proto { tcp, udp } to $nameservers port domain
pass in on $ext_if proto tcp to $webserver port $webports
pass in on $int_if proto tcp from $localnet to $webserver port $webports
pass out on $dmz_if proto tcp to $webserver port $webports
pass in log on $ext_if proto tcp to $mailserver port smtp
pass in log on $int_if proto tcp from $localnet to $mailserver \
     port $email
pass out log on $dmz_if proto tcp to $mailserver port smtp
pass in on $dmz_if from $mailserver to port smtp
pass out log on $ext_if proto tcp from $mailserver to port smtp
```

You could choose to make the other `pass` rules that reference your local network interface-specific, too, but if you leave them intact, they'll continue to work.

Sharing the Load: Redirecting to a Pool of Addresses

Once you've set up services to be accessible to the world at large, one likely scenario is that over time, one or more of your services will grow more sophisticated and resource-hungry or simply attract more traffic than you feel comfortable serving from a single server.

There are a number of ways to make several machines share the load of running a service, including ways to fine-tune the service itself. For the network-level load balancing, PF offers the basic functionality you need via redirection to tables or address pools. In fact, you can implement a form of load balancing without even touching your `pass` rules, at least if your environment is not yet dual-stack.

Take the Web server in our example. We already have the macro that represents a service, our Web server. For reasons that will become obvious in a moment, we need to reduce that macro to represent only the public IPv4 address (`webserver = "192.0.2.227"`), which, in turn, is associated with the hostname that your users have bookmarked, possibly *www.example.com*. When the time comes to share the load, set up the required number of identical, or at least equivalent, servers and then alter your rule set slightly to introduce the redirection. First, define a table that holds the addresses for your Web server pool's IPv4 addresses:

```
table <webpool> persist { 192.0.2.214, 192.0.2.215, 192.0.2.216, 192.0.2.217 }
```

Then, perform the redirection:

```
match in on $ext_if protp tcp to $webserver port $webports \
        rdr-to <webpool> round-robin
```

Unlike the redirections in earlier examples, such as the FTP proxy in Chapter 3, this rule sets up all members of the `webpool` table as potential redirection targets for incoming connections intended for the `webports` ports on the `webserver` address. Each incoming connection that matches this rule is redirected to one of the addresses in the table, spreading the load across several hosts. You may choose to retire the original Web server once the switch to this redirection is complete, or you may let it be absorbed in the new Web server pool.

On PF versions earlier than OpenBSD 4.7, the equivalent rule is as follows:

```
rdr on $ext_if proto tcp to $webserver port $webports -> <webpool> round-robin
```

In both cases, the `round-robin` option means that PF shares the load between the machines in the pool by cycling through the table of redirection addresses sequentially.

Some applications expect accesses from each individual source address to always go to the same host in the backend (for example, there are services that depend on client- or session-specific parameters that will be lost if new connections hit a different host in the backend). If your configuration needs to cater to such services, you can add the `sticky-address` option to make sure that new connections from a client are always redirected to the same machine behind the redirection as the initial connection. The downside to this option is that PF needs to maintain source-tracking data for each client, and the default value for maximum source nodes tracked is set at 10,000, which may be a limiting factor. (See Chapter 10 for advice on adjusting this and similar limit values.)

When even load distribution isn't an absolute requirement, selecting the redirection address at `random` may be appropriate:

```
match in on $ext_if proto tcp to $webserver port $webports \
        rdr-to <webpool> random
```

NOTE *On pre–OpenBSD 4.7 PF versions, the `random` option isn't supported for redirection to tables or lists of addresses.*

Even organizations with large pools of official, routable IPv4 addresses have opted to introduce NAT between their load-balanced server pools and the Internet at large. This technique works equally well in various NAT-based setups, but moving to NAT offers some additional possibilities and challenges.

In order to accommodate an IPv4 and IPv6 dual-stack environment in this way, you'll need to set up separate tables for address pools and separate `pass` or `match` rules with redirections for IPv4 and IPv6. A single table of both IPv4 and IPv6 addresses may sound like an elegant idea at first, but the simple redirection rules outlined here aren't intelligent enough to make correct redirection decisions based on the address family of individual table entries.

Getting Load Balancing Right with relayd

After you've been running for a while with load balancing via round-robin redirection, you may notice that the redirection doesn't automatically adapt to external conditions. For example, unless special steps are taken, if a host in the list of redirection targets goes down, traffic will still be redirected to the IP addresses in the list of possibilities.

Clearly, a monitoring solution is needed. Fortunately, the OpenBSD base system provides one. The relay daemon `relayd`[2] interacts with your PF configuration, providing the ability to weed out nonfunctioning hosts from your pool. Introducing `relayd` into your setup, however, may require some minor changes to your rule set.

The `relayd` daemon works in terms of two main classes of services that it refers to as *redirects* and *relays*. It expects to be able to add or subtract hosts' IP addresses to or from the PF tables it controls. The daemon interacts

2. Originally introduced in OpenBSD 4.1 under the name `hoststated`, the daemon has seen active development (mainly by Reyk Floeter and Pierre-Yves Ritschard) over several years, including a few important changes to the configuration syntax, and it was renamed `relayd` in time for the OpenBSD 4.3 release.

with your rule set through a special-purpose anchor named relayd (and in pre–OpenBSD 4.7 versions, also a redirection anchor, rdr-anchor, with the same name).

To see how we can make our sample configuration work a little better by using relayd, we'll look back at the load-balancing rule set. Starting from the top of your *pf.conf* file, add the anchor for relayd to insert rules as needed:

```
anchor "relayd/*"
```

On pre–OpenBSD 4.7 versions, you also need the redirection anchor:

```
rdr-anchor "relayd/*"
anchor "relayd/*"
```

In the load-balancing rule set, we had the following definition for our Web server pool:

```
table webpool persist { 192.0.2.214, 192.0.2.215, 192.0.2.216, 192.0.2.217 }
```

It has this match rule to set up the redirection:

```
match in on $ext_if proto tcp to $webserver port $webports \
       rdr-to <webpool> round-robin
```

Or on pre–OpenBSD 4.7 versions, you'd use the following:

```
rdr on $ext_if proto tcp to $webserver port $webports -> <webpool> round-robin
```

To make this configuration work slightly better, we remove the redirection and the table (remember to take care of both sets in a dual-stack configuration), and we let relayd handle the redirection or redirections by setting up its own versions inside the anchor. (Don't remove the pass rule, however, because your rule set will still need to have a pass rule that lets traffic flow to the IP addresses in relayd's tables. If you had separate rules for your inet and inet6 traffic, you may be able to merge those rules back into one.)

Once the *pf.conf* parts have been taken care of, we turn to relayd's own *relayd.conf* configuration file. The syntax in this configuration file is similar enough to *pf.conf* to make it fairly easy to read and understand. First, we add the macro definitions we'll be using later:

```
web1="192.0.2.214"
web2="192.0.2.215"
web3="192.0.2.216"
web4="192.0.2.217"
webserver="192.0.2.227"
sorry_server="192.0.2.200"
```

All of these correspond to definitions we could have put in a *pf.conf* file. The default checking interval in relayd is 10 seconds, which means that a host could be down for almost 10 seconds before it's taken offline. Being cautious, we'll set the checking interval to 5 seconds to minimize visible downtime, with the following line:

```
interval 5 # check hosts every 5 seconds
```

Now, we make a table called webpool that uses most of the macros:

```
table <webpool> { $web1, $web2, $web3, $web4 }
```

For reasons we'll return to shortly, we define one other table:

```
table <sorry> { $sorry_server }
```

At this point, we're ready to set up the redirect:

```
redirect www {
       listen on $webserver port 80 sticky-address
       tag relayd
       forward to <webpool> check http "/status.html" code 200 timeout 300
       forward to <sorry> timeout 300 check icmp
}
```

This redirect says that connections to port 80 should be redirected to the members of the webpool table. The sticky-address option has the same effect here as the rdr-to in PF rules: New connections from the same source IP address (within the time interval defined by the timeout value) are redirected to the same host in the backend pool as the previous ones.

The relayd daemon should check to see whether a host is available by asking it for the file */status.html*, using the protocol HTTP, and expecting the return code to be equal to 200. This is the expected result for a client asking a running Web server for a file it has available.

No big surprises so far, right? The relayd daemon will take care of excluding hosts from the table if they go down. But what if all the hosts in the webpool table go down? Fortunately, the developers thought of that, too, and introduced the concept of backup tables for services. This is the last part of the definition for the www service, with the table sorry as the backup table: The hosts in the sorry table take over if the webpool table becomes empty. This means that you need to configure a service that's able to offer a "Sorry, we're down" message in case all the hosts in your webpool fail.

If you're running an IPv6-only service, you should, of course, substitute your IPv6 addresses for the ones given in the example earlier. If you're running a dual-stack setup, you should probably set up the load-balancing mechanism separately for each protocol, where the configurations differ only in names (append a 4 or 6, for example, to the IPv4 and IPv6 sets of names, respectively) and the addresses themselves.

With all of the elements of a valid relayd configuration in place, you can enable your new configuration.

Before you actually start relayd, add an empty set of relayd_flags to your */etc/rc.conf.local* to enable:

```
relayd_flags="" # for normal use: ""
```

Reload your PF rule set and then start relayd. If you want to check your configuration before actually starting relayd, you can use the -n command-line option to relayd:

```
$ sudo relayd -n
```

If your configuration is correct, relayd displays the message configuration OK and exits.

To actually start the daemon, you could start relayd without any command-line flags, but as with most daemons, it's better to start it via its rc script wrapper stored in */etc/rc.d/*, so the following sequence reloads your edited PF configuration and enables relayd.

```
$ sudo pfctl -f /etc/pf.conf
$ sudo sh /etc/rc.d/relayd start
```

With a correct configuration, both commands will silently start, without displaying any messages. (If you prefer more verbose messages, both pfctl and relayd offer the -v flag. For relayd, you may want to add the -v flag to the *rc.conf.local* entry.) You can check that relayd is running with top or ps. In both cases, you'll find three relayd processes, roughly like this:

```
$ ps waux | grep relayd
_relayd   9153  0.0  0.1   776  1424 ??  S      7:28PM    0:00.01 relayd: pf update engine
(relayd)
_relayd   6144  0.0  0.1   776  1440 ??  S      7:28PM    0:00.02 relayd: host check engine
(relayd)
root      3217  0.0  0.1   776  1416 ??  Is     7:28PM    0:00.01 relayd: parent (relayd)
```

And as we mentioned earlier, with an empty set of relayd_flags in your *rc.conf.local* file, relayd is enabled at startup. However, once the configuration is enabled, most of your interaction with relayd will happen through the relayctl administration program. In addition to letting you monitor status, relayctl lets you reload the relayd configuration and selectively disable or enable hosts, tables, and services. You can even view service status interactively, as follows:

```
$ sudo relayctl show summary
Id      Type            Name                            Avlblty Status
1       redirect        www                                     active
1       table           webpool:80                              active (2 hosts)
1       host            192.0.2.214                     100.00% up
2       host            192.0.2.215                     0.00%   down
```

```
3       host            192.0.2.216                     100.00% up
4       host            192.0.2.217                     0.00%   down
2       table           sorry:80                                active (1 hosts)
5       host            127.0.0.1                       100.00% up
```

In this example, the `webpool` is seriously degraded, with only two of four hosts up and running. Fortunately, the backup table is still functioning, and hopefully it'll still be up if the last two servers fail as well. For now, all tables are active with at least one host up. For tables that no longer have any members, the `Status` column changes to empty. Asking `relayctl` for host information shows the status information in a host-centered format:

```
$ sudo relayctl show hosts
Id      Type            Name                            Avlblty Status
1       table           webpool:80                              active (3 hosts)
1       host            192.0.2.214                     100.00% up
                        total: 11340/11340 checks
2       host            192.0.2.215                     0.00%   down
                        total: 0/11340 checks, error: tcp connect failed
3       host            192.0.2.216                     100.00% up
                        total: 11340/11340 checks
4       host            192.0.2.217                     0.00%   down
                        total: 0/11340 checks, error: tcp connect failed
2       table           sorry:80                                active (1 hosts)
5       host            127.0.0.1                       100.00% up
                        total: 11340/11340 checks
```

If you need to take a host out of the pool for maintenance (or any time-consuming operation), you can use `relayctl` to disable it, as follows:

```
$ sudo relayctl host disable 192.0.2.217
```

In most cases, the operation will display `command succeeded` to indicate that the operation completed successfully. Once you've completed maintenance and put the machine online, you can reenable it as part of `relayd`'s pool with this command:

```
$ sudo relayctl host enable 192.0.2.217
```

Again, you should see the message `command succeeded` almost immediately to indicate that the operation was successful.

In addition to the basic load balancing demonstrated here, `relayd` has been extended in recent OpenBSD versions to offer several features that make it attractive in more complex settings. For example, it can now handle Layer 7 proxying or relaying functions for HTTP and HTTPS, including protocol handling with header append and rewrite, URL-path append and rewrite, and even session and cookie handling. The protocol handling needs to be tailored to your application. For example, the following is a simple HTTPS relay for load balancing the encrypted Web traffic from clients to the Web servers.

```
http protocol "httpssl" {
        header append "$REMOTE_ADDR" to "X-Forwarded-For"
        header append "$SERVER_ADDR:$SERVER_PORT" to "X-Forwarded-By"
        header change "Keep-Alive" to "$TIMEOUT"
        query hash "sessid"
        cookie hash "sessid"
        path filter "*command=*" from "/cgi-bin/index.cgi"

        ssl { sslv2, ciphers "MEDIUM:HIGH" }
        tcp { nodelay, sack, socket buffer 65536, backlog 128 }
}
```

This protocol handler definition demonstrates a range of simple operations on the HTTP headers and sets both SSL parameters and specific TCP parameters to optimize connection handling. The header options operate on the protocol headers, inserting the values of the variables by either appending to existing headers (`append`) or changing the content to a new value (`change`).

The URL and cookie hashes are used by the load balancer to select to which host in the target pool the request is forwarded. The `path filter` specifies that any get request, including the first quoted string as a substring of the second, is to be dropped. The `ssl` options specify that only SSL version 2 ciphers are accepted, with key lengths in the medium-to-high range—in other words, 128 bits or more.[3] Finally, the `tcp` options specify `nodelay` to minimize delays, specify the use of the selective acknowledgment method (RFC 2018), and set the socket buffer size and the maximum allowed number of pending connections the load balancer keeps track of. These options are examples only; in most cases, your application will perform well with these settings at their default values.

The relay definition using the protocol handler follows a pattern that should be familiar given the earlier definition of the `www` service:

```
relay wwwssl {
        # Run as a SSL accelerator
        listen on $webserver port 443 ssl
        protocol "httpssl"
        table <webhosts> loadbalance check ssl
}
```

Still, your SSL-enabled Web applications will likely benefit from a slightly different set of parameters.

NOTE *We've added a `check ssl`, assuming that each member of the `webhosts` table is properly configured to complete an SSL handshake. Depending on your application, it may be useful to look into keeping all SSL processing in `relayd`, thus offloading the encryption-handling tasks from the backends.*

3. See the OpenSSL man page for further explanation of cipher-related options.

Finally, for CARP-based failover of the hosts running `relayd` on your network (see Chapter 8 for information about CARP), `relayd` can be configured to support CARP interaction by setting the CARP demotion counter for the specified interface groups at shutdown or startup.

Like all others parts of the OpenBSD system, `relayd` comes with informative man pages. For the angles and options not covered here (there are a few), dive into the man pages for `relayd`, `relayd.conf`, and `relayctl` and start experimenting to find just the configuration you need.

A Web Server and Mail Server on the Inside—The NAT Version

Let's backtrack a little and begin again with the baseline scenario where the sample clients from Chapter 3 get three new neighbors: a mail server, a Web server, and a file server. This time around, externally visible IPv4 addresses are either not available or too expensive, and running several other services on a machine that's primarily a firewall isn't desirable. This means we're back to the situation where we do our NAT at the gateway. Fortunately, the redirection mechanisms in PF make it relatively easy to keep servers on the inside of a gateway that performs NAT.

The network specifications are the same as for the *example.com* setup we just worked through: We need to run a Web server that serves up data in cleartext (`http`) and encrypted (`https`) form, and we want a mail server that sends and receives email while letting clients inside and outside the local network use a number of well-known submission and retrieval protocols. In short, we want pretty much the same features as in the setup from the previous section, but with only one routable address.

Of the three servers, only the Web server and the mail server need to be visible to the outside world, so we add macros for their IP addresses and services to the Chapter 3 rule set:

```
webserver = "192.168.2.7"
webports = "{ http, https }"
emailserver = "192.168.2.5"
email = "{ smtp, pop3, imap, imap3, imaps, pop3s }"
```

With only one routable address and the servers hidden in NATed address space, we need to set up rules at the gateway that redirect the traffic we want our servers to handle. We could define a set of `match` rules to set up the redirection and then address the `block` or `pass` question in a separate set of rules later, like this:

```
match in on $ext_if proto tcp to $ext_if port $webports rdr-to $webserver
match in on $ext_if proto tcp to $ext_if port $email rdr-to $emailserver

pass proto tcp to $webserver port $webports
pass proto tcp to $emailserver port $email
pass proto tcp from $emailserver to port smtp
```

This combination of match and pass rules is very close to the way things were done in pre–OpenBSD 4.7 PF versions, and if you're upgrading from a previous version, this is the kind of quick edit that could bridge the syntax gap quickly. But you could also opt to go for the new style and write this slightly more compact version instead:

```
pass in on $ext_if inet proto tcp to $ext_if port $webports rdr-to $webserver tag RDR
pass in on $ext_if inet proto tcp to $ext_if port $email rdr-to $mailserver tag RDR
pass on $int_if inet tagged RDR
```

Note the use of pass rules with rdr-to. This combination of filtering and redirection will help make things easier in a little while, so try this combination for now.

On pre–OpenBSD 4.7 PF, the rule set will be quite similar, except in the way that we handle the redirections.

```
webserver = "192.168.2.7"
webports = "{ http, https }"
emailserver = "192.168.2.5"
email = "{ smtp, pop3, imap, imap3, imaps, pop3s }"

rdr on $ext_if proto tcp to $ext_if port $webports -> $webserver
rdr on $ext_if proto tcp to $ext_if port $email -> $emailserver

pass proto tcp to $webserver port $webports
pass proto tcp to $emailserver port $email
pass proto tcp from $emailserver to any port smtp
```

DMZ with NAT

With an all-NAT setup, the pool of available addresses to allocate for a DMZ is likely to be larger than in our previous example, but the same principles apply. When you move the servers off to a physically separate network, you'll need to check that your rule set's macro definitions are sane and adjust the values if necessary.

Just as in the routable-addresses case, it might be useful to tighten up your rule set by editing your pass rules so the traffic to and from your servers is allowed to pass on only the interfaces that are actually relevant to the services:

```
pass in on $ext_if inet proto tcp to $ext_if port $webports rdr-to $webserver
pass in on $int_if inet proto tcp from $localnet to $webserver port $webports
pass out on $dmz_if proto tcp to $webserver port $webports
pass in log on $ext_if inet proto tcp to $ext_if port $email rdr-to $mailserver
pass in log on $int_if proto tcp from $localnet to $mailserver port $email
pass out log on $dmz_if proto tcp to $mailserver port smtp
pass in on $dmz_if from $mailserver to port smtp
pass out log on $ext_if proto tcp from $mailserver to port smtp
```

The version for pre–OpenBSD 4.7 PF differs in some details, with the redirection still in separate rules:

```
pass in on $ext_if proto tcp to $webserver port $webports
pass in on $int_if proto tcp from $localnet to $webserver port $webports
pass out on $dmz_if proto tcp to $webserver port $webports
pass in log on $ext_if proto tcp to $mailserver port smtp
pass in log on $int_if proto tcp from $localnet to $mailserver port $email
pass out log on $dmz_if proto tcp to $mailserver port smtp
pass in on $dmz_if from $mailserver to port smtp
pass out log on $ext_if proto tcp from $mailserver to port smtp
```

You could create specific pass rules that reference your local network interface, but if you leave the existing pass rules intact, they'll continue to work.

Redirection for Load Balancing

The redirection-based load-balancing rules from the previous example work equally well in a NAT regime, where the public address is the gateway's external interface and the redirection addresses are in a private range.

Here's the webpool definition:

```
table <webpool> persist { 192.168.2.7, 192.168.2.8, 192.168.2.9, 192.168.2.10 }
```

The main difference between the routable-address case and the NAT version is that after you've added the webpool definition, you edit the existing pass rule with redirection, which then becomes this:

```
pass in on $ext_if inet proto tcp to $ext_if port $webports rdr-to <webpool> round-robin
```

Or for pre–OpenBSD 4.7 PF versions, use this:

```
rdr on $ext_if proto tcp to $ext_if port $webports -> <webpool> round-robin
```

From that point on, your NATed DMZ behaves much like the one with official, routable addresses.

NOTE *You can configure a valid IPv6 setup to coexist with a NATed IPv4 setup like this one, but if you choose to do so, be sure to treat* inet *and* inet6 *traffic separately in your PF rules. And contrary to popular belief, rules with* nat-to *and* rdr-to *options work in IPv6 configurations the same as in IPv4.*

Back to the Single NATed Network

It may surprise you to hear that there are cases where setting up a small network is more difficult than working with a large one. For example, returning to the situation where the servers are on the same physical network as the clients, the basic NATed configuration works very well—up to a point. In fact, everything works brilliantly as long as all you're interested in is getting traffic from hosts outside your local network to reach your servers.

Here's the full configuration:

```
ext_if = "re0" # macro for external interface - use tun0 or pppoe0 for PPPoE
int_if = "re1" # macro for internal interface
localnet = $int_if:network
# for ftp-proxy
proxy = "127.0.0.1"
icmp_types = "{ echoreq, unreach }"
client_out = "{ ssh, domain, pop3, auth, nntp, http, https, \
                446, cvspserver, 2628, 5999, 8000, 8080 }"
udp_services = "{ domain, ntp }"
webserver = "192.168.2.7"
webports = "{ http, https }"
emailserver = "192.168.2.5"
email = "{ smtp, pop3, imap, imap3, imaps, pop3s }"
# NAT: ext_if IP address could be dynamic, hence ($ext_if)
match out on $ext_if from $localnet nat-to ($ext_if)
block all
# for ftp-proxy: Remember to put the following line, uncommented, in your
# /etc/rc.conf.local to enable ftp-proxy:
# ftpproxy_flags=""
anchor "ftp-proxy/*"
pass in quick proto tcp to port ftp rdr-to $proxy port 8021
pass out proto tcp from $proxy to port ftp
pass quick inet proto { tcp, udp } to port $udp_services
pass proto tcp to port $client_out
# allow out the default range for traceroute(8):
# "base+nhops*nqueries-1" (33434+64*3-1)
pass out on $ext_if inet proto udp to port 33433 >< 33626 keep state
# make sure icmp passes unfettered
pass inet proto icmp icmp-type $icmp_types from $localnet
pass inet proto icmp icmp-type $icmp_types to $ext_if
pass in on $ext_if inet proto tcp to $ext_if port $webports rdr-to $webserver
pass in on $ext_if inet proto tcp to $ext_if port $email rdr-to $mailserver
pass on $int_if inet proto tcp to $webserver port $webports
pass on $int_if inet proto tcp to $mailserver port $email
```

The last four rules here are the ones that interest us the most. If you try to reach the services on the official address from hosts in your own network, you'll soon see that the requests for the redirected services from machines in your local network most likely never reach the external interface. This is because all the redirection and translation happens on the external interface. The gateway receives the packets from your local network on the internal interface, with the destination address set to the external interface's address. The gateway recognizes the address as one of its own and tries to handle the request as if it were directed at a local service; as a consequence, the redirections don't quite work from the inside.

The equivalent part to those last four lines of the preceding rule set for pre–OpenBSD 4.7 systems looks like this:

```
rdr on $ext_if proto tcp to $ext_if port  $webports -> $webserver
rdr on $ext_if proto tcp to $ext_if port  $email -> $emailserver

pass proto tcp to $webserver port $webports
pass proto tcp to $emailserver port $email
pass proto tcp from $emailserver to any port smtp
```

Fortunately, several work-arounds for this particular problem are possible. The problem is common enough that the PF User Guide lists four different solutions to the problem,[4] including moving your servers to a DMZ, as described earlier. Because this is a PF book, we'll concentrate on a PF-based solution (actually a pretty terrible work-around), which consists of treating the local network as a special case for our redirection and NAT rules.

We need to intercept the network packets originating in the local network and handle those connections correctly, making sure that any return traffic is directed to the communication partner who actually originated the connection. This means that in order for the redirections to work as expected from the local network, we need to add special-case redirection rules that mirror the ones designed to handle requests from the outside. First, here are the `pass` rules with redirections for OpenBSD 4.7 and newer:

```
pass in on $ext_if inet proto tcp to $ext_if port $webports rdr-to $webserver
pass in on $ext_if inet proto tcp to $ext_if port $email rdr-to $mailserver
pass in log on $int_if inet proto tcp from $int_if:network to $ext_if \
port $webports rdr-to $webserver
pass in log on $int_if inet proto tcp from $int_if:network to $ext_if \
port $email rdr-to $mailserver
match out log on $int_if proto tcp from $int_if:network to $webserver \
port $webports nat-to $int_if
pass on $int_if inet proto tcp to $webserver port $webports
match out log on $int_if proto tcp from $int_if:network to $mailserver \
port $email nat-to $int_if
pass on $int_if inet proto tcp to $mailserver port $email
```

The first two rules are identical to the original ones. The next two intercept the traffic from the local network, and the `rdr-to` actions in both rewrite the destination address, much as the corresponding rules do for the traffic that originates elsewhere. The `pass on $int_if` rules serve the same purpose as in the earlier version.

The `match` rules with `nat-to` are there as a routing work-around. Without them, the `webserver` and `mailserver` hosts would route return traffic for the redirected connections directly back to the hosts in the local network, where the traffic wouldn't match any outgoing connection. With the `nat-to` in

4. See the "Redirection and Reflection" section in the PF User Guide (*http://www.openbsd.org/faq/pf/rdr.html#reflect*).

place, the servers consider the gateway as the source of the traffic and will direct return traffic back the same path it came originally. The gateway matches the return traffic to the states created by connections from the clients in the local network and applies the appropriate actions to return the traffic to the correct clients.

The equivalent rules for pre–OpenBSD 4.7 versions are at first sight a bit more confusing, but the end result is the same.

```
rdr on $int_if proto tcp from $localnet to $ext_if port $webports -> $webserver
rdr on $int_if proto tcp from $localnet to $ext_if port $email -> $emailserver
no nat on $int_if proto tcp from $int_if to $localnet
nat on $int_if proto tcp from $localnet to $webserver port $webports -> $int_if
nat on $int_if proto tcp from $localnet to $emailserver port $email -> $int_if
```

This way, we twist the redirections and the address translation logic to do what we need, and we don't need to touch the `pass` rules at all. (I've had the good fortune to witness via email and IRC the reactions of several network admins at the moment when the truth about this five-line reconfiguration sank in.)

Filtering on Interface Groups

Your network could have several subnets that may never need to interact with your local network except for some common services, like email, Web, file, and print. How you handle the traffic from and to such subnets depends on how your network is designed. One useful approach is to treat each less-privileged network as a separate local network attached to its own separate interface on a common filtering gateway and then to give it a rule set that allows only the desired direct interaction with the neighboring networks attached to the main gateway.

You can make your PF configuration more manageable and readable by grouping logically similar interfaces into interface groups and by applying filtering rules to the groups rather than the individual interfaces. Interface groups, as implemented via the `ifconfig` *`group`* option, originally appeared in OpenBSD 3.6 and have been adopted in FreeBSD 7.0 onward.

All configured network interfaces can be configured to belong to one or more groups. Some interfaces automatically belong to one of the default groups. For example, all IEEE 802.11 wireless network interfaces belong to the `wlan` group, while interfaces associated with the default routes belong to the `egress` group. Fortunately, an interface can be a member of several groups, and you can add interfaces to interface groups via the appropriate `ifconfig` command, as in this example:

```
# ifconfig sis2 group untrusted
```

For a permanent configuration, the equivalent under OpenBSD would be in the *hostname.sis2* file or the `ifconfig_sis2=` line in the *rc.conf* file on FreeBSD 7.0 or later.

Where it makes sense, you can then treat the interface group much the same as you would handle a single interface in filtering rules:

```
pass in on untrusted to any port $webports
pass out on egress to any port $webports
```

If by now you're thinking that in most, if not all, the rule-set examples up to this point, it would be possible to filter on the group `egress` instead of the macro `$ext_if`, you've grasped an important point. It could be a useful exercise to go through any existing rule sets you have and see what using interface groups can do to help readability even further. Remember that an interface group can have one or more members.

Note that filtering on interface groups makes it possible to write essentially hardware-independent rule sets. As long as your *hostname.if* files or `ifconfig_if=` lines put the interfaces in the correct groups, rule sets that consistently filter on interface groups will be fully portable between machines that may or may not have identical hardware configurations.

On systems where the interface group feature isn't available, you may be able to achieve some of the same effects via creative use of macros, as follows:

```
untrusted = "{ ath0 ath1 wi0 ep0 }"
egress = "sk0"
```

The Power of Tags

In some networks, the decision of where a packet should be allowed to pass can't be made to map easily to criteria like subnet and service. The fine-grained control the site's policy demands could make the rule set complicated and potentially hard to maintain.

Fortunately, PF offers yet another mechanism for classification and filtering in the form of *packet tagging*. The useful way to implement packet tagging is to `tag` incoming packets that match a specific `pass` rule and then let the packets pass elsewhere based on which identifiers the packet is `tagged` with. In OpenBSD 4.6 and later, it's even possible to have separate `match` rules that `tag` according to the match criteria, leaving decisions on passing, redirecting, or taking other actions to rules later in the rule set.

One example could be the wireless access points we set up in Chapter 4, which we could reasonably expect to inject traffic into the local network with an apparent source address equal to the access point's `$ext_if` address. In that scenario, a useful addition to the rule set of a gateway with several of these access points might be the following (assuming, of course, that definitions of the `wifi_allowed` and `wifi_ports` macros fit the site's requirements):

```
wifi = "{ 10.0.0.115, 10.0.0.125, 10.0.0.135, 10.0.0.145 }"
pass in on $int_if from $wifi to $wifi_allowed port $wifi_ports tag wifigood
pass out on $ext_if tagged wifigood
```

As the complexity of the rule set grows, consider using `tag` in incoming `match` and `pass` rules to make your rule set readable and easier to maintain.

Tags are sticky, and once a packet has been tagged by a matching rule, the tag stays, which means that a packet can have a tag even if it wasn't applied by the last matching rule. However, a packet can have only one tag at any time. If a packet matches several rules that apply tags, the tag will be overwritten with a new one by each new matching `tag` rule.

For example, you could set several tags on incoming traffic via a set of `match` or `pass` rules, supplemented by a set of `pass` rules that determine where packets pass out based on the tags set on the incoming traffic.

The Bridging Firewall

An Ethernet *bridge* consists of two or more interfaces that are configured to forward Ethernet frames transparently and that aren't directly visible to the upper layers, such as the TCP/IP stack. In a filtering context, the bridge configuration is often considered attractive because it means that the filtering can be performed on a machine that doesn't have its own IP addresses. If the machine in question runs OpenBSD or a similarly capable operating system, it can still filter and redirect traffic.

The main advantage of such a setup is that attacking the firewall itself is more difficult.[5] The disadvantage is that all admin tasks must be performed at the firewall's console, unless you configure a network interface that's reachable via a secured network of some kind or even a serial console. It also follows that bridges with no IP address configured can't be set as the gateway for a network and can't run any services on the bridged interfaces. Rather, you can think of a bridge as an intelligent bulge on the network cable, which can filter and redirect.

A few general caveats apply to using firewalls implemented as bridges:

- The interfaces are placed in promiscuous mode, which means that they'll receive (and to some extent process) every packet on the network.
- Bridges operate on the Ethernet level and, by default, forward all types of packets, not just TCP/IP.
- The lack of IP addresses on the interfaces makes some of the more effective redundancy features, such as CARP, unavailable.

The method for configuring bridges differs among operating systems in some details. The following examples are very basic and don't cover all possible wrinkles, but they should be enough to get you started.

5. How much security this actually adds is a matter of occasional heated debate on mailing lists such as *openbsd-misc* and other networking-oriented lists. Reading up on the pros and cons as perceived by core OpenBSD developers can be entertaining as well as enlightening.

Basic Bridge Setup on OpenBSD

The OpenBSD GENERIC kernel contains all the necessary code to configure bridges and filter on them. Unless you've compiled a custom kernel without the bridge code, the setup is quite straightforward.

NOTE *On OpenBSD 4.7 and newer, the `brconfig` command no longer exists. All bridge configuration and related functionality was merged into `ifconfig` for the OpenBSD 4.7 release. If you're running on an OpenBSD release where `brconfig` is available, you're running an out-of-date, unsupported configuration. Please upgrade to a more recent version as soon as feasible.*

To set up a bridge with two interfaces on the command line, you first create the bridge device. The first device of a kind is conventionally given the sequence number 0, so we create the `bridge0` device with the following command:

```
$ sudo ifconfig bridge0 create
```

Before the next `ifconfig` command, use `ifconfig` to check that the prospective member interfaces (in our case, `ep0` and `ep1`) are up, but not assigned IP addresses. Next, configure the bridge by entering the following:

```
$ sudo ifconfig bridge0 add ep0 add ep1 blocknonip ep0 blocknonip ep1 up
```

The OpenBSD `ifconfig` command contains a fair bit of filtering code itself. In this example, we use the `blocknonip` option for each interface to block all non-IP traffic.

NOTE *The OpenBSD `ifconfig` command offers its own set of filtering options in addition to other configuration options. The `bridge(4)` and `ifconfig(8)` man pages provide further information. Because it operates on the Ethernet level, it's possible to use `ifconfig` to specify filtering rules that let the bridge filter on MAC addresses. Using these filtering capabilities, it's also possible to let the bridge tag packets for further processing in your PF rule set via the `tagged` keyword. For tagging purposes, a bridge with one member interface will do.*

To make the configuration permanent, create or edit */etc/hostname.ep0* and enter the following line:

```
up
```

For the other interface, */etc/hostname.ep1* should contain the same line:

```
up
```

Finally, enter the bridge setup in */etc/hostname.bridge0*:

```
add ep0 add ep1 blocknonip ep0 blocknonip ep1 up
```

Your bridge should now be up, and you can go on to create the PF filter rules.

Basic Bridge Setup on FreeBSD

For FreeBSD, the procedure is a little more involved than on OpenBSD. In order to be able to use bridging, your running kernel must include the `if_bridge` module. The default kernel configurations build this module, so under ordinary circumstances, you can go directly to creating the interface. To compile the bridge device into the kernel, add the following line in the kernel configuration file:

```
device if_bridge
```

You can also load the device at boot time by putting the following line in the */etc/loader.conf* file.

```
if_bridge_load="YES"
```

Create the bridge by entering this:

```
$ sudo ifconfig bridge0 create
```

Creating the `bridge0` interface also creates a set of bridge-related `sysctl` values:

```
$ sudo sysctl net.link.bridge
net.link.bridge.ipfw: 0
net.link.bridge.pfil_member: 1
net.link.bridge.pfil_bridge: 1
net.link.bridge.ipfw_arp: 0
net.link.bridge.pfil_onlyip: 1
```

It's worth checking that these `sysctl` values are available. If they are, it's confirmation that the bridge has been enabled. If they're not, go back and see what went wrong and why.

NOTE *These values apply to filtering on the bridge interface itself. You don't need to touch them because IP-level filtering on the member interfaces (the ends of the pipe) is enabled by default.*

Before the next `ifconfig` command, check that the prospective member interfaces (in our case, `ep0` and `ep1`) are up but haven't been assigned IP addresses. Then configure the bridge by entering this:

```
$ sudo ifconfig bridge0 addm ep0 addm ep1 up
```

To make the configuration permanent, add the following lines to */etc/rc.conf*:

```
ifconfig_ep0="up"
ifconfig_ep1="up"
cloned_interfaces="bridge0"
ifconfig_bridge0="addm ep0 addm ep1 up"
```

This means your bridge is up and you can go on to create the PF filter rules. See the `if_bridge(4)` man page for further FreeBSD-specific bridge information.

Basic Bridge Setup on NetBSD

On NetBSD, the default kernel configuration doesn't have the filtering bridge support compiled in. You need to compile a custom kernel with the following option added to the kernel configuration file. Once you have the new kernel with the bridge code in place, the setup is straightforward.

```
options         BRIDGE_IPF      # bridge uses IP/IPv6 pfil hooks too
```

To create a bridge with two interfaces on the command line, first create the `bridge0` device:

```
$ sudo ifconfig bridge0 create
```

Before the next `brconfig` command, use `ifconfig` to check that the prospective member interfaces (in our case, `ep0` and `ep1`) are up but haven't been assigned IP addresses. Then, configure the bridge by entering this:

```
$ sudo brconfig bridge0 add ep0 add ep1 up
```

Next, enable the filtering on the `bridge0` device:

```
$ sudo brconfig bridge0 ipf
```

To make the configuration permanent, create or edit */etc/ifconfig.ep0* and enter the following line:

```
up
```

For the other interface, */etc/ifconfig.ep1* should contain the same line:

```
up
```

Finally, enter the bridge setup in */etc/ifconfig.bridge0*:

```
create
!add ep0 add ep1 up
```

Your bridge should now be up, and you can go on to create the PF filter rules. For further information, see the PF on NetBSD documentation at *http://www.netbsd.org/Documentation/network/pf.html*.

The Bridge Rule Set

Figure 5-3 shows the *pf.conf* file for a bulge-in-the-wire version of the baseline rule set we started in this chapter. As you can see, the network changes slightly.

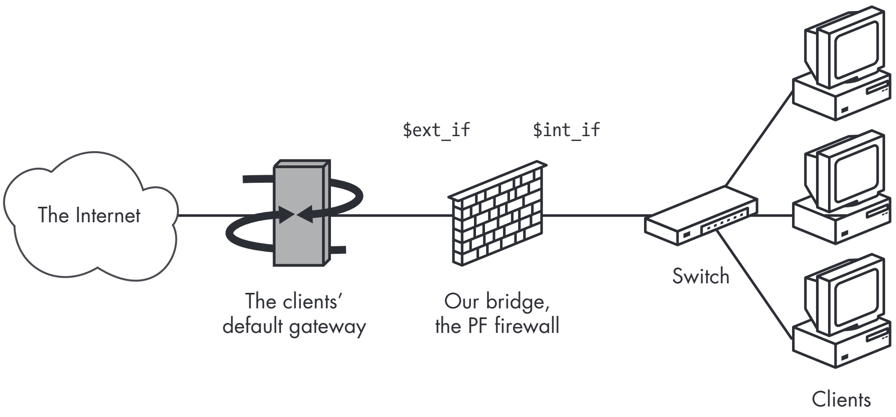

Figure 5-3: A network with a bridge firewall

The machines in the local network share a common default gateway, which isn't the bridge but could be placed either inside or outside the bridge.

```
ext_if = ep0
int_if  = ep1
localnet= "192.0.2.0/24"
webserver = "192.0.2.227"
webports = "{ http, https }"
emailserver = "192.0.2.225"
email = "{ smtp, pop3, imap, imap3, imaps, pop3s }"
nameservers = "{ 192.0.2.221, 192.0.2.223 }"
client_out = "{ ssh, domain, pop3, auth, nntp, http, https, \
                446, cvspserver, 2628, 5999, 8000, 8080 }"
udp_services = "{ domain, ntp }"
icmp_types = "{ echoreq, unreach }"
set skip on $int_if
block all
pass quick on $ext_if inet proto { tcp, udp } from $localnet \
        to port $udp_services
pass log on $ext_if inet proto icmp all icmp-type $icmp_types
pass on $ext_if inet proto tcp from $localnet to port $client_out
pass on $ext_if inet proto { tcp, udp } to $nameservers port domain
pass on $ext_if proto tcp to $webserver port $webports
pass log on $ext_if proto tcp to $emailserver port $email
pass log on $ext_if proto tcp from $emailserver to port smtp
```

Significantly more complicated setups are possible. But remember that while redirections will work, you won't be able to run services on any of the interfaces without IP addresses.

Handling Nonroutable IPv4 Addresses from Elsewhere

Even with a properly configured gateway to handle filtering and potentially NAT for your own network, you may find yourself in the unenviable position of needing to compensate for other people's misconfigurations.

Establishing Global Rules

One depressingly common class of misconfigurations is the kind that lets traffic with nonroutable addresses out to the Internet. Traffic from nonroutable IPv4 addresses plays a part in several *denial-of-service (DoS)* attack techniques, so it's worth considering explicitly blocking traffic from nonroutable addresses from entering your network. One possible solution is outlined here. For good measure, it also blocks any attempt to initiate contact to nonroutable addresses through the gateway's external interface.

```
martians = "{ 127.0.0.0/8, 192.168.0.0/16, 172.16.0.0/12, \
              10.0.0.0/8, 169.254.0.0/16, 192.0.2.0/24, \
              0.0.0.0/8, 240.0.0.0/4 }"

block in quick on $ext_if from $martians to any
block out quick on $ext_if from any to $martians
```

Here, the `martians` macro denotes the RFC 1918 addresses and a few other ranges mandated by various RFCs not to be in circulation on the open Internet. Traffic to and from such addresses is quietly dropped on the gateway's external interface.

NOTE *The* `martians` *macro could easily be implemented as a table instead, with all of the table advantages as an added bonus for your rule set. In fact, if you view the loaded rules in a rule set that contains this combination of macro and rules, you'll see that macro expansion and rule-set optimization most likely replaced your list with one table per rule. However, if you roll your own table, you'll get to pick a nicer name for it yourself.*

The specific details of how to implement this kind of protection will vary according to your network configuration and may be part of a wider set of network security measures. Your network design might also dictate that you include or exclude address ranges other than these.

Restructuring Your Rule Set with Anchors

We've mentioned anchors a few times already, in the context of applications such as FTP-proxy or `relayd` that use anchors to interact with a running PF configuration. Anchors are named sub–rule sets where it's possible to insert or remove rules as needed without reloading the whole rule set.

Once you have a rule set where an otherwise unused anchor is defined, you can even manipulate anchor contents from the command line using `pfctl`'s `-a` switch, like this:

```
echo "block drop all" | pfctl -a baddies -f -
```

Here, a rule is inserted into the existing anchor `baddies`, overwriting any previous content.

You can even load rules from a separate file into an anchor:

```
pfctl -a baddies -f /etc/anchor-baddies
```

Or you can list the current contents of an anchor:

```
pfctl -a baddies -s rules
```

NOTE *There are a few more `pfctl` options that you'll find useful for handling anchors. See the `pfctl` man page for inspiration.*

You can also split your configuration by putting the contents of anchors into separate files to be loaded at rule-set load time. That way it becomes possible to edit the rules in the anchors separately, reload the edited anchor, and, of course, do any other manipulation like the ones described above. To do this, first add a line like this to *pf.conf*:

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

This references the file */etc/anchor-ssh-good*, which could look like this:

```
table <sshbuddies> file "/etc/sshbuddies"
     pass inet proto tcp from <sshbuddies> to any port ssh
```

Perhaps simply to make it possible to delegate the responsibility for the table `sshbuddies` to a junior admin, the anchor loads the table from the file */etc/sshbuddies*, which could look like this:

```
192.168.103.84
10.11.12.13
```

This way, you can manipulate the contents of the anchor in the following ways: Add rules by editing the file and reloading the anchor, replace the rules by feeding other rules from the command line via standard input (as shown in the earlier example), or change the behavior of the rules inside the anchor by manipulating the contents of the table they reference.

NOTE *For more extensive anchors, like the ones discussed in the following paragraphs, it's probably more useful to use `include` clauses in your* pf.conf *if you want to maintain the anchors as separate files.*

The concept hinted at previously (specifying a set of common criteria that apply to all actions within an anchor) is appealing in situations where your configuration is large enough to need a few extra structuring aids. For example, "on interface" could be a useful common criterion for traffic arriving on a specific interface because that traffic tends to have certain similarities. For example, look at the following:

```
anchor "dmz" on $dmz_if {
    pass in proto { tcp udp } to $nameservers port domain
    pass in proto tcp to $webservers port { www https }
    pass in proto tcp to $mailserver port smtp
    pass in log (all, to pflog1) in proto tcp from $mailserver \
                to any port smtp
    }
```

A separate anchor ext would serve the egress interface group:

```
anchor ext on egress {
    match out proto tcp to port { www https } set queue (qweb, qpri) set prio (5,6)
    match out proto { tcp udp } to port domain set queue (qdns, qpri) set prio (6,7)
    match out proto icmp set queue (q_dns, q_pri) set prio (7,6)
    pass in log proto tcp to port smtp rdr-to 127.0.0.1 port spamd queue spamd
    pass in log proto tcp from <nospamd> to port smtp
    pass in log proto tcp from <spamd-white> to port smtp
    pass out log proto tcp to port smtp
    pass log (all) proto { tcp, udp } to port ssh keep state (max-src-conn 15, \
        max-src-conn-rate 7/3, overload <bruteforce> flush global)
}
```

Another obvious logical optimization if you group rules in anchors based on interface affinity is to lump in tags to help policy-routing decisions. A simple but effective example could look like this:

```
anchor "dmz" on $dmz_if {
    pass in proto { tcp udp } to $nameservers port domain tag GOOD
    pass in proto tcp to $webservers port { www https } tag GOOD
    pass in proto tcp to $mailserver port smtp tag GOOD
    pass in log (all, to pflog1) in proto tcp from $mailserver
                to any port smtp tag GOOD
    block log quick ! tagged GOOD
    }
```

Even if the anchor examples here have all included a blocking decision inside the anchor, the decision to block or pass based on tag information doesn't have to happen inside the anchor.

After this whirlwind tour of anchors as a structuring tool, it may be tempting to try to convert your entire rule set to an anchors-based structure. If you try to do so, you'll probably find ways to make the internal logic clearer. But don't be surprised if certain rules need to be global, outside of anchors tied to common criteria. And you'll almost certainly find that what turns out to be useful in your environment is at least a little different from what inspired the scenarios I've presented here.

How Complicated Is Your Network?—Revisited

Early on in this chapter, we posed the questions "How complicated is your network?" and "How complicated does it need to be?" Over the subsections of this chapter, we've presented a number of tools and techniques that make it possible to build complex infrastructure with PF and related tools and that help manage that complexity while keeping the network administrator sane.

If you're in charge of one site where you need to apply all or most of the techniques we've mentioned in this chapter, I feel your pain. On the other hand, if you're in charge of a network that diverse, the subsequent chapters on traffic shaping and managing resource availability are likely to be useful to you as well.

The rest of this book deals mainly with optimizing your setup for performance and resource availability, with the exception of one chapter where we deviate slightly and take on a lighter tone. Before we dive into how to optimize performance and ensure high availability, it's time to take a look at how to make your infrastructure unavailable or hard to reach for selected groups or individuals. The next chapter deals exclusively with making life harder for the unwashed masses—or perhaps even well-organized criminals—who try to abuse services in your care.