Firewalling with OpenBSD’s PF packet filter

Peter N. M. Hansteen

peter@bsdly.net

The document is a 'work in progress', based on a manuscript prepared for a lecture at the BLUG (see http://www.blug.linux.no/) meeting of January 27th, 2005.

I'm interested in comments of all kinds, and you may if you wish add web or other references to html or pdf versions of the manuscript. If you do, I would like, but can not require, you to send me an email message that you've done it. For communication regarding this document please use the address <peter@bsdly.net>; **whois bsdly.net** provides full contact information. *For full revision history, see the HTML version or the SGML source.*

Revision History

Revision 0.09651e 27 april 2007
typokill edition
Revision 0.09655e 13 may 2007
BSDCan 2007 edition
Revision 0.096551e 28 may 2007
typofix. Thanks: Austin Hook. While here, update references
Revision 0.0966e 11 sep 2007
EuroBSDCon 2007 edition

Table of Contents

Before we start **1**
PF? **3**
Packet filter? Firewall? **5**
NAT? **6**
PF today **8**
BSD vs Linux - Configuration **9**
Simplest possible setup (OpenBSD) **10**
Simplest possible setup (FreeBSD) **12**
Simplest possible setup (NetBSD) **13**
First rule set - single machine **14**
Slightly stricter **15**
Statistics from pfctl **17**
A simple gateway, NAT if you need it **19**
Gateways and the pitfalls of in, out and on 19
What is your local network, anyway? 20
Setting up 21
That sad old FTP thing **25**
FTP through NAT: ftp-proxy 25
FTP, PF and routable addresses: ftpsesame, pftpx and ftp-proxy! 27
ftp-proxy, new style 28
Making your network troubleshooting friendly **30**
Then, do we let it all through? 30
The easy way out: The buck stops here 31
Letting ping through 31
Helping traceroute 32
Path MTU discovery 32
Network hygiene: Blocking, scrubbing and so on **34**
`block-policy` 34
`scrub` 34
`antispoof` 34
Handling non-routable addresses from elsewhere 35
A web server and a mail server on the inside **37**
Taking care of your own - the inside 37

Tables make your life easier...40
Logging...42
  Taking a peek with tcpdump 42
  Other log tools you may want to look into...............................43
  But there are limits (an anecdote)...44
Keeping an eye on things with pftop...45
Invisible gateway - bridge...46
Directing traffic with ALTQ...48
  ALTQ - prioritizing by traffic type...49
    So why does this work? 49
  ALTQ - allocation by percentage 50
  ALTQ - handling unwanted traffic 51
CARP and pfsync...53
Wireless networks made simple 54
  A little IEEE 802.11 background..54
    WEP (Wired Equivalent Privacy).......................................54
    WPA (WiFi Protected Access) 55
  Setting up a simple wireless network 55
An open, yet tightly guarded wireless network with authpf............58
Turning away the brutes..62
  expiretable tidies your tables...64
Giving spammers a hard time..66
  Remember, you are not alone: blacklisting 66
  List of black and grey, and the sticky tarpit 66
  Setting up spamd...67
  Some early highlights of our spamd experience 69
  Beating'em up some more: spamdb and greytrapping 72
    Enter greytrapping 73
    Your own traplist..73
    Deleting, handling trapped entries 74
    The downside: some people really do not get it 75
  Conclusions from our spamd experience....................................76
PF - Haiku..78
References 79
Where to find the tutorial on the web..81
  If you enjoyed this: Buy OpenBSD CDs and other items, donate! 81

Before we start

This lecture[1] will be about firewalls and related functions, starting from *a little* theory along with a number of examples of filtering and other network traffic directing. As in any number of other endeavors, the things I discuss can be done *in more than one way*. Under any circumstances I will urge you to interrupt me when you need to. That is, if you will permit me to use what I learn from your comments later, either in revised versions of this lecture or in practice at a later time. But first,

This is not a HOWTO

This document is not intended as a pre-cooked recipe for cutting and pasting.

Just to hammer this in, please repeat after me

```
The Pledge of the Network Admin

This is my network.

It is mine
or technically my employer’s,
it is my responsibility
and I care for it with all my heart

there are many other networks a lot like mine,

but none are just like it.

I solemnly swear

that I will not mindlessly paste from HOWTOs.
```

[1] This manuscript is a slightly further developed version of a manuscript prepared for a lecture which was announced as (translated from Norwegian): "This lecture is about firewalls and related functions, with examples from real life with the OpenBSD project’s PF (Packet Filter). PF offers firewalling, NAT, traffic control and bandwidth management in a single, flexible and sysadmin friendly system. Peter hopes that the lecture will give you some ideas about how to control your network traffic the way you want - keeping some things outside your network, directing traffic to specified hosts or services, and of course, giving spammers a hard time."
People who have offered significant and useful input regarding this manuscript include Eystein Roll Aarseth, David Snyder, Peter Postma, Henrik Kramshøj, Vegard Engen, Greg Lehey, Ian Darwin, Daniel Hartmeier, Mark Uemura, Hallvor Engen and probably a few who will remain lost in my mail archive until I can grep them out of there.
I would like to thank the following organizations for their kind support: The NUUG Foundation for a travel grant which partly financed my AUUG2005 appearance; The AUUG, UKUUG, SANE, BSDCan, EuroBSDCon and AsiaBSDCon organizations for inviting me to their conferences; and finally the FreeBSD Foundation for sponsoring my trips to BSDCan 2006 and EuroBSDCon 2006.

> The point is, while the rules and configurations I show you do work, I have tested them and they are in some way related to what has been put into production, they may very well be overly simplistic and are almost certain to be at least a little off and possibly quite wrong for *your* network.
>
> Please keep in mind that this document is intended to show you a few useful things and inspire you to achieve good things.
>
> Please strive to understand your network and what you need to do to make it better.
>
> Please do not paste blindly from this document or any other.

Now, with that out of the way, we can go on to the meat of the matter.

PF?

What, then is PF? Let us start by looking briefly at the project's history to put things in their proper context.

OpenBSD's Packet Filter subsystem, which most people refer to simply by using the abbreviated form 'PF', was originally written in an effort of extremely rapid development during the northern hemisphere summer and autumn months of 2001 by Daniel Hartmeier and a number of OpenBSD developers, and was launched as a default part of the OpenBSD 3.0 base system in December of 2001.

The need for a new firewalling software subsystem for OpenBSD arose when Darren Reed announced to the world that IPFilter, which at that point had been rather intimately integrated in OpenBSD, was not after all BSD licensed. In fact quite to the contrary. The license itself was almost a word by word copy of the BSD license, omitting only the right to make changes to the code and distribute the result. The OpenBSD version of IPFilter contained quite a number of changes and customizations, which it turned out were not allowed according to the license. IPFilter was removed from the OpenBSD source tree on May 29th, 2001, and for a few weeks OpenBSD-current did not contain any firewalling software.

Fortunately, in Switzerland Daniel Hartmeier was already doing some limited experiments involving kernel hacking in the networking code.

His starting point was hooking a small function of his own into the networking stack, making packets pass through it, and after a while he had started thinking about filtering. Then the license crisis happened.

IPFilter was pruned from the source tree on May 29th. The first commit of the PF code happened Sunday, June 24 2001 at 19:48:58 UTC.[1]

A few months of rather intense activity followed, and the version of PF to be released with OpenBSD 3.0 contained a rather complete implementation of packet filtering, including network address translation.

[1] It is worth noting that the IPFilter copyright episode spurred the OpenBSD team to perform a *license audit* of the entire source tree and ports in order to avoid similar situations in the future. A number of potential problems were uncovered and resolved over the months that followed, removing a number of potetial license pitfalls for everyone involved in free software development. Theo de Raadt summed up the effort in a message to the openbsd-misc mailing list on February 20th, 2003, available among others from the MARC mailing list archives (http://marc.info/?l=openbsd-misc&m=104570938124454&w=2).

From the looks of it, Daniel Hartmeier and the other PF developers made good use of their experience with the IPFilter code. Under any circumstances Daniel presented a USENIX 2002 paper with performance tests which show that the OpenBSD 3.1 PF performed equally well as or better under stress than IPFilter on the same platform or iptables on Linux.

In addition, some tests were run on the original PF from OpenBSD 3.0. These tests showed mainly that the code had gained in efficiency from version 3.0 to version 3.1. The article which provides the details is available from Daniel Hartmeier's web, see *http://www.benzedrine.cx/pf-paper.html*.

I have not seen comparable tests performed recently, but in my own experience and that of others, the PF filtering overhead is pretty much negligible. As one data point, the machine which gateways between Datadok's network and the world is a Pentium III 450MHz with 384MB of RAM. When I've remembered to check, I've never seen the machine at less than 96 percent 'idle' according to top.

Packet filter? Firewall?

By now I have already used some terms and concepts before I've bothered to explain them, and I'll correct that oversight shortly. PF is a *packet filter*, that is, code which inspects network packets at the protocol and port level, and decides what to do with them. In PF's case this code for the most part operates in kernel space, inside the network code.

PF operates in a world which consists of packets, protocols, connections and ports.

Based on where a packet is coming from or where it's going, which protocol, connection or port it is designated for, PF is able to determine where to lead the packet, or decide if it is to be let through at all.

It's equally possible to direct network traffic based on packet *contents*, usually referred to as application level filtering, but this is not the kind of thing PF does. We will come back later to some cases where PF will hand off these kinds of tasks to other software, but first let us deal with some basics.

We've already mentioned the firewall concept. One important feature of PF and similar software, perhaps the most important feature, is that it is able to identify and block traffic which you do not want to let into your local network or let out to the world outside. At some point the term '*firewall*' was coined.

While blocking "bad" traffic and denying access can be quite important, I tend to emphasize the somewhat wider and more general perspective that the packet filter is a very flexible tool which is extremely useful when you want to *take control* of what happens in your network.

Taking control means you get to make informed decisions, and that, in my opinion, is when the fun part of being a network administrator starts. And you should be forewarned, staying in control is a recurring theme in this session.

NAT?

One other thing we will be talking about quite a lot are ’inner’ and ’outer’ addresses, ’routable’ and ’non-routable’ adresses. This is, at the heart of things, not directly related to firewalls or packet filtering, but due to the way the world works today, we will need to touch on it.

It all comes from the time in the early 1990s when somebody started calculating the number of computers which would be connected to the Internet if the commercialization continued and the great unwashed masses of consumers were to connect at the same time.

At the time the Internet protocols were formulated, computers were usually big, expensive things which would normally serve a large number of simultaneous users, each at their own more or less dumb terminal. Under any circumstances, the only ones allowed to connect were universities and a number of companies with Pentagon contracts. Essentially 32 bit addresses of 4 octets would go an extremely long way. It would accommodate literally millions of machines, even.

Then Internet commercialization happened, and all of a sudden there were actually millions of small, inexpensive machines wanting to connect at the same time. This kind of development showed every sign of continuing and even accelerating. This meant that the smart people who had made the net work, needed to do another few pieces of work. They did a few things more or less at the same time. For one, they started working on a solution based on a larger address space - this has been dubbed IP version 6, or IPv6 for short - which uses 128 bit addresses. This has been designated as the long term solution. I thought I’d mention at this point that IPv6 support is built into OpenBSD by default, and PF has as far as I know always contained IPv6 support[1].

In addition, some sort of temporary solution was needed. Making the world move to addresses four times the size would take considerable time. The process is as far as we can see still pretty much in an early stage. They found a temporary solution, which consists of two parts. One part was a mechanism to offer the rest of the world ’white lies’ by letting the network gateways rewrite packet addresses, the other was offered by designating some address ranges which had not been assigned earlier for use only in

[1] But IPv6 adoption is still in the early stages, and as more people are starting to look at the protocols in earnest, issues are discovered, some with potentially serious security implications such as the ICMP6 design flaw which generated some controversy in early 2007.

networks which would not communicate directly with the Internet at large. This would mean that several different machines at separate locations could have the same local IP address. But this would not matter because the address would be translated before the traffic was let out to the net at large.

If traffic with such "non routable" addresses were to hit the Internet at large, routers seeing the traffic would have a valid reason to refuse the packets to pass any further.

This is what is called "Network Address Translation", sometimes referred to as "IP masquerade" or similar. The two RFCs which define the whats and hows of this are dated 1994 and 1996 respectively [2].

There may be a number of reasons to use the so called "RFC 1918 addresses", but traditionally and historically the main reason has been that official addresses are either not available or practical.

[2] The two documents are RFC 1631, "The IP Network Address Translator (NAT)", dated May 1994 and RFC 1918, "Address Allocation for Private Internets", dated February 1996. See the chapter called *References*

PF today

At this point, we have covered a bit of background. Some years have passed since 2001, and PF in its present OpenBSD 4.2 form is a packet filter which is capable of doing quite a few things, if you want it to.

For one thing, PF classifies packets based on protocol, port, packet type, source or destination address. With a reasonable degree of certainty it is also able to classify packets based on source operating system.

And even if NAT is not a necessary part of a packet filter, for practical reasons it's nice that the address rewriting logic is handled somewhere nearby. Consequently, PF contains NAT logic as well.

PF is able - based on various combinations of protocol, port and other data - to direct traffic to other destinations than those designated by the sender, for example to a different machine or for further processing by a program such as a daemon listening at a port, locally or on a different machine.

Before PF was written, OpenBSD already contained the *ALTQ* code to handle load balancing and traffic shaping. After a while, altq was integrated with PF. Mainly for practical reasons.

As a result of this, all those features are available to you via one single, essentially human readable configuration file, which is usually called `pf.conf`, stored in the `/etc/` directory.

This is now available as a part of the base system on OpenBSD, on FreeBSD where PF from version 5.3 is one of three firewalling systems to be loaded at will, and in NetBSD and DragonFlyBSD. The last two operating systems I have not had the resources to play much with myself. Something about having both hardware and time available at the same time. Anyway all indications are that only very minor details vary between these systems as far as PF is involved.

BSD vs Linux - Configuration

I assume that some of those who read this document are more familiar with configuring Linux or other systems than with BSD, so I'll briefly mention a few points about BSD system and network configuration.

BSD network interfaces are not labeled `eth0`, `eth1` and so on. The interfaces are assigned names which equal the driver name plus a sequence number, making 3Com cards using the `xl` driver appear as `xl0`, `xl1`, and so on, while Some Intel cards are likely to end up as `em0`, `em1`, others are supported by the `fxp` driver, and so on. There may even be slight variations in which cards are supported in which drivers across the BSDs.[1]

For boot-time configuration, the BSDs are generally organized to read the configuration from `/etc/rc.conf`, which is read by the `/etc/rc` script at startup. OpenBSD recommends using `/etc/rc.conf.local` for local customizations, since `rc.conf` contains the default values, while FreeBSD uses `/etc/defaults/rc.conf` to store the default settings, making `/etc/rc.conf` the correct place to make changes.

PF is configured by editing the `/etc/pf.conf` file and by using the pfctl command line tool. The pfctl application has a *large* number of options. We will take a closer look at some of them today.

In case you are wondering, there are web interfaces available for admin tasks, but they are not parts of the base system. The PF developers are not hostile to these things, but rather have not seen any graphical interface to PF configuration which without a doubt is preferable to `pf.conf` in a text editor, backed up with pfctl invocations and a few unix tricks.

[1] When in doubt, consult the output of the dmesg command, which displays the kernel message buffer. Under most circumstances, the kernel's hardware probing and recognition messages will be intact in the message buffer for a relatively long time after your system has finished booting.

Simplest possible setup (OpenBSD)

This brings us, finally, to the practical point of actually configuring PF in the simplest possible setup. We'll deal with a single machine which will communicate with a network which may very well be the Internet.

In order to start PF, as previously mentioned, you need to tell the rc system that you want the service to start. On OpenBSD, this is done in `/etc/rc.conf.local`, with the magical line

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

quite simply. In addition, you may if you like specify the file where PF will find its rules.

```
pf_rules=/etc/pf.conf # specify which file contains your rules
```

The default value is the one given here, `/etc/pf.conf`. At the next startup, PF will be enabled. You can verify this by looking for the message `PF enabled` on the console. The `/etc/pf.conf` which comes out of a normal install of OpenBSD, FreeBSD or NetBSD contains a number of useful suggestions, but they're all commented out.

Then again, you really do not need to restart your machine in order to enable PF. You can do this just as easily by using pfctl. We really do not want to reboot for no good reason, so we type the command

```
peter@skapet:~$ sudo pfctl -ef /etc/pf.conf
```

which enables PF and loads your rule set.[1][2]. At this point we do not have a rule set, which means that PF does not actually do anything.

It is probably worth noting that if you reboot your machine at this point, the rc script on OpenBSD at least will enable a default rule set, which is in fact loaded *before* any of the network interfaces are enabled.

This default rule set is designed as a safety measure in case your gateway boots with an invalid configuration. It lets you log in and clean up

[1] As a footnoted aside, I tend to use sudo when I need to do something which requires privileges. Sudo is in the base system on OpenBSD, and is within easy reach as a port or package elsewhere. If you have not started using sudo yet, you should. Then you'll avoid shooting your own foot simply because you forgot you were root in that terminal window.

[2] For convenience if you want it - pfctl is able to handle several operations on a single command line. You can, for example, enable PF and load the rule set with the command **`sudo pfctl -ef /etc/pf.conf`**, and bunch on several other options if you like.

whichever syntax error caused your rule set not to load. The default rule set allows a basic set of services: ssh from anyhere, basic name resolution and NFS mounts.

Some early versions of PF ports elsewhere appear to have neglected to bring the default rules with them.

Simplest possible setup (FreeBSD)

On FreeBSD it could seem that you need a little more magic in your `/etc/rc.conf`, specifically according to the FreeBSD Handbook[1]

```
pf_enable="YES"                  # Enable PF (load module if required)
pf_rules="/etc/pf.conf"          # rules definition file for PF
pf_flags=""                      # additional flags for pfctl startup
pflog_enable="YES"               # start pflogd(8)
pflog_logfile="/var/log/pflog"   # where pflogd should store the logfile
pflog_flags=""                   # additional flags for pflogd startup
```

Fortunately almost all of these are already present as the default settings in your `/etc/defaults/rc.conf`, and only

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

are in fact needed as additions to your `/etc/rc.conf` in order to enable PF.

On FreeBSD, PF by default is compiled as a kernel loadable module. This means that you should be able to get started[2] right after you have added those two lines to your configuration with `$` **`sudo kldload pf`**, followed by `$` **`sudo pfctl -e`** to enable PF. Assuming you have put these lines in your `/etc/rc.conf`, you can use the PF rc script to enable or disable PF:

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

to enable PF, or

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

to disable the packet filter. The `pf` rcNG script supports a few other operations as well. However it is still worth noting that at this point we do not have a rule set, which means that PF does not actually do anything.

[1] There are some differences between the FreeBSD 4.n and 5.n and newer releases with respect to PF. Refer to the FreeBSD Handbook, specifically the PF chapter (http://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/firewalls-pf.html) to see which information applies in your case.

[2] Here I use the **sudo** command, which is an excellent tool when you need to do something which requires privileges. **sudo** is not part of the base system on FreeBSD, but is within easy reach from the ports system as `security/sudo`. If you have not started using sudo yet, you should. Then you'll avoid shooting your own foot simply because you forgot you were root in that terminal window.

Simplest possible setup (NetBSD)

On NetBSD 2.0 and newer PF is available as a loadable kernel module, installed via packages as `pkgsrc/security/pflkm` or compiled into a static kernel configuration. In NetBSD 3.0 onwards, PF is part of the base system.

If you want to enable PF in your kernel configuration (rather than loading the kernel module), you add these lines to your kernel configuration:

```
pseudo-device  pf # PF packet filter
pseudo-device  pflog # PF log interface
```

In `/etc/rc.conf` you need the lines

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

to enable loadable kernel modules, PF and the PF log interface, respectively.

If you installed the module, you load it with `NetBSD$` **`sudo modload /usr/lkm/pf.o`**, followed by `NetBSD$` **`sudo pfctl -e`** to enable PF. Alternatively, you can run the rc scripts, `NetBSD$` **`sudo /etc/rc.d/pf start`** to enable PF and `NetBSD$` **`sudo /etc/rc.d/pflogd start`** to enable the logging.

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

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

If it's still all correct at this point, you are ready to create your first PF rule set.

First rule set - single machine

This is the simplest possible setup, for a single machine which will not run any services, and which will talk to one network which may be the Internet. For now, we will use a `/etc/pf.conf` which looks like this:

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

that is, deny any incoming traffic, allow traffic we make ourselves, and retain state information on our connections. Keeping state information allows return traffic for all connections we have initiated to pass back to us. It is worth noting that from OpenBSD 4.1 onwards, the default for pass rules is to keep state information[1], so the equivalent rule set in the new OpenBSD 4.1 style is even simpler,

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

It goes pretty much without saying that passing all traffic generated by a specific host implies a great deal of trust that the host in question is, in fact, trustworthy. This is something you do if and only if this is a machine you know you can trust. If you are ready to use the rule set, you load it with

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

[1] 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

Slightly stricter

For a slightly more structured and complete setup, we start by denying everything and then allowing only those things we know that we need[1]. This gives us the opportunity to introduce two of the features which make PF such a wonderful tool - lists and macros.

We'll make some changes to `/etc/pf.conf`, starting with

```
block all
```

Then we back up a little. Macros need to be defined before use:

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

Now we've demonstrated several things at once - what macros look like, we've shown that macros may be lists, and that PF understands rules using port names equally well as it does port numbers. The names are the ones listed in `/etc/services`. This gives us something to put in our rules, which we edit slightly to look like this:

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

Please remember to add **`keep state`** to these rules if your system has a PF version older than OpenBSD 4.1.

At this point some of us will point out that UDP is stateless, but PF actually manages to maintain state information despite this. When you ask a name server about a domain name, it is reasonable to assume that you probably want to receive the answer. Retaining state information about your UDP traffic achieves this.

Since we've made changes to our `pf.conf` file, we load the new rules:

```
peter@skapet:~$ sudo pfctl -f /etc/pf.conf
```

[1] You may ask why do I write the rule set to default deny? The short answer is, it gives you better control at the expense of some thinking. The point of packet filtering is to take control, not to run catch-up with what the bad guys do. Marcus Ranum has written a very entertaining and informative article about this, The Six Dumbest Ideas in Computer Security (http://www.ranum.com/security/computer_security/editorials/dumb/index.html), which comes highly recommended. It is a good read.

and the new rules apply. If there are no syntax errors, pfctl will not output any messages during the rule load. The -v flag will produce more verbose pfctl output.

If you have made extensive changes to your rule set, you may want to check the rules before attempting to load them. The command to do this is, **`pfctl -nf`** `/etc/pf.conf`. The **`-n`** option causes the rules to be interpreted only without loading the rules. This gives you an opportunity to correct any errors. Under any circumstances the last valid rule set loaded will be in force until you either disable PF or load a new rule set.

That is worth noting: When loading a new rule set, the last valid rule set stays loaded until the new one is fully parsed and loaded, and PF switches directly from one to the other. There is no intermediate stage with no rules loaded or a mixture of the two rule sets.

Statistics from pfctl

You may want to check that PF is actually running, and perhaps at the same time look at some statistics. The pfctl program offers a number of different types of information if you use **pfctl -s**, adding the type of information you want to display. The following example is taken from my home gateway while I was preparing an earlier version of this lecture:

```
peter@skapet:~$ sudo pfctl -s info
Status: Enabled for 17 days 00:24:58          Debug: Urgent

Interface Stats for ep0               IPv4             IPv6
  Bytes In                      9257508558                0
  Bytes Out                      551145119              352
  Packets In
    Passed                         7004355                0
    Blocked                          18975                0
  Packets Out
    Passed                         5222502                3
    Blocked                             65                2

State Table                          Total             Rate
  current entries                       15
  searches                        19620603           13.3/s
  inserts                           173104            0.1/s
  removals                          173089            0.1/s
Counters
  match                             196723            0.1/s
  bad-offset                             0            0.0/s
  fragment                              22            0.0/s
  short                                  0            0.0/s
  normalize                              0            0.0/s
  memory                                 0            0.0/s
  bad-timestamp                          0            0.0/s
  congestion                             0            0.0/s
  ip-option                             28            0.0/s
  proto-cksum                          325            0.0/s
  state-mismatch                       983            0.0/s
  state-insert                           0            0.0/s
  state-limit                            0            0.0/s
  src-limit                             26            0.0/s
  synproxy                               0            0.0/s
```

The first line here indicates that PF is enabled and has been running for for a little more than two weeks, which is equal to the time since I upgraded to what was then the latest snapshot. **pfctl -s all** provides highly detailed information. Try it and have a look, and while there, look into some of the other pfctl options. **`man 8 pfctl`** gives you full information.

At this point you have a single machine which should be able to communicate reasonably well with other internet connected machines. And while the rule set is very basic, it serves as an excellent starting point for staying in control of your network.

This is a very basic rule set and a few things are still missing. For example, you probably want to let at least some ICMP and UDP traffic through, if nothing else for your own troubleshooting needs.

And even though more modern and more secure options are available, you will probably be required to handle the ftp service.

We will return to these items shortly.

A simple gateway, NAT if you need it

At this point we finally move on to the more realistic or at least more common setups, where the machine with the packet filter or firewall configured also acts as a gateway for at least one other machine.

The other machines on the inside may of course also run firewall software, but even if they do, it does not affect what we are interested in here to any significant degree.

Gateways and the pitfalls of in, out and on

In the single machine setup, life is relatively simple. Traffic you create should either pass or not out to the rest of the world, and you decide what you let in from elsewhere.

When you set up a gateway, your perspective changes. You go from the "me versus the network out there" setting to "I am the one who decides what to pass to or from all the networks I am connected to". The machine has several, or at least two, network interfaces, each connected to a separate net.

Now it's very reasonable to think that if you want traffic to pass from the network connected to **ep1** to hosts on the network connected to **ep0**, you will need a rule like

```
pass in inet proto tcp on ep1 from ep1:network to ep0:network \
     port $ports keep state
```

which keeps track of states as well.[1]

However, one of the most common and most complained-about mistakes in firewall configuration is not realizing that the "to" keyword does not in itself guarantee passage all the way there. In fact, the rule we just wrote only lets the traffic pass in to the gateway itself, on the specific interface given in the rule.

[1] In fact, even if the **keep state** part denotes the default behaviour and is redundant if you are working with OpenBSD 4.1 or equivalent, there is no need to remove the specification from your rules when upgrading your rule set from earlier versions. To ease transition, the examples in this tutorial will make this distinction when needed.

To let the packets get a bit further on and move into the next network over, you would need a matching rule which says something like

```
pass out inet proto tcp on ep0 from ep1:network to ep0:network \
     port $ports keep state
```

These rules will work, but they will not necessarily achieve what you want.

If there are good reasons why you need to have rules which are this specific in your rule set, you know you need them and why. By the end of this tutorial you should be able to articulate with some precision, in the context of your own network, just when such rules are needed. However for the basic gateway configurations we'll be dealing with here, what you really want to use is probably a rule which says

```
pass inet proto tcp from ep1:network to any port $ports keep state
```

to let your local net access the Internet and leave the detective work to the *antispoof* and *scrub* code. They are both pretty good these days, and we will get back to them later. For now we just accept the fact that for simple setups, interface bound rules with in/out rules tend to add more clutter than they are worth to your rule sets.

For a busy network admin, a readable rule set is a safer rule set.

For the remainder of this tutorial, with some exceptions, we will keep the rules as simple as possible for readability.

What is your local network, anyway?

Above we introduced the **`interface:network`** notation. That is a nice piece of shorthand, but you make your rule set even more readable and maintainable by taking the macro use a tiny bit further.

You could for example define a **`$localnet`** macro, initially defined as the network directly attached to your internal interface (**`ep1:network`** in the examples above).

Alternatively, 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 **`fec0:dead:beef::/64`** for an IPv6 range.

If your network requires it, you could even define your **`$localnet`** as a list of networks. Whatever your specific needs, a sensible **`$localnet`** definition and a typical pass rule of the type

```
pass inet proto tcp from $localnet to any port $ports keep state
```

could end up saving you a few headaches. We will stick to that convention from here on.

Setting up

We assume that the machine has acquired another network card or at any rate you have set up a network connection from your local network, via PPP or other means. We will not consider the specific interface configurations.

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

First, we need to turn on gatewaying in order to let the machine forward the network traffic it receives on one interface to other networks via a separate interface. Initially we will do this on the command line with sysctl, for traditional *IP version four*

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

and if we need to forward *IP version six* traffic, the command is

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

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`, by changing the lines you need, like this

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

On FreeBSD, you conventionally do the corresponding 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 the sysctl command. However, a larger part of the FreeBSD configuration is centralized into the `rc.conf` file.

Are both of the interfaces you intend to use up and running? Use **`ifconfig -a`**, or **`ifconfig`** *interface_name* to find out.

If you still intend to allow any traffic initiated by machines on the inside, your `/etc/pf.conf` could look roughly like this[2]:

```
ext_if = "ep0" # macro for external interface - use tun0 for PPPoE
int_if = "ep1"  # macro for internal interface
localnet = $int_if:network
# ext_if IP address could be dynamic, hence ($ext_if)
nat on $ext_if from $localnet to any -> ($ext_if)
block all
pass inet proto tcp from { lo0, $localnet } to any keep state
```

Note the use of macros to assign logical names to the network interfaces. Here 3Com cards are used, but this is the last time during this lecture we will find this of any interest whatsoever. In truly simple setups like this one, we may not gain very much by using macros like these, but once the rule sets grow somewhat larger, you will learn to appreciate the readability this adds to the rule sets

Also note the nat rule. This is where we handle the network address translation from the non-routable address inside your local net to the sole official address we assume has been assigned to you.

The parentheses surrounding the last part of the nat rule **`($ext_if)`** serve 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 external IP address changes.

On the other hand, this rule set probably allows more traffic than what you actually want to pass out of your network. Where I work, the equivalent macro is

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

[2] For dialup users who conventionally use some variant of PPP for their Internet connections, the external interface is the `tun0` pseudo-device. Broadband users such as ADSL subscribers tend to have an Ethernet interface to play with, however for a significant subset of ADSL users, specifically those using PPP over Ethernet (PPPoE), the correct external interface will be the `tun0` pseudo-device, not the physical Ethernet interface.

with the rule

```
pass inet proto tcp from $localnet to any port $client_out \
     flags S/SA keep state
```

This may be a somewhat peculiar selection of ports, but it's exactly what my colleagues and I need. Some of the numbered ports are needed for systems I am not allowed to discuss any further. Your needs probably differ at least in some specifics, but this should cover at least some of the more useful services.

In addition, we have a few other pass rules. We will be returning to some of the more interesting ones rather soon. One pass rule which is useful to those of us who want the ability to administer our machines from elsewhere is

```
pass in inet proto tcp from any to any port ssh
```

or for that matter

```
pass in inet proto tcp from any to $ext_if port ssh
```

whichever you like. Lastly we need to make the name service and time keeping work for our clients

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

supplemented with a rule which passes the traffic we want through our firewall:

```
pass quick inet proto { tcp, udp } to any port $udp_services keep state
```

Note the **quick** keyword in this rule. We have started writing rule sets which consist of several rules, and it is time to take a look at the relationships between the rules in a rule set. The rules are evaluated from top to bottom, in the sequence they are written in the configuration file. For each packet or connection evaluated by PF, *the last matching rule* in the rule set is the one which is 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 which might have matched the packet. Quite handy when you need a few isolated exceptions to your general rules.

It is worth noting that we used one rule for both the domain name service (**`domain`** and time synchronization (**`ntp`**). The most important reason for this is that both protocols under certain circumstances communicate alternately over TCP and UDP.

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. FTP is 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 not like. The most common points against it, are

- Passwords are transferred in the clear
- 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 selected at random

All of these points make for challenges security-wise, even before considering any potential weaknesses in client or server software which may lead to security issues. These things have tended to happen.

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

Regardless of our professionalism and preferences, we are all too aware that at times we will need to handle things we would prefer not to. In the case of FTP through firewalls, the main part of our handling consists of redirecting the traffic to a small program which is written specifically for this purpose.

Depending on your configuration, which operating system you are using as the platform for your PF firewall and how you count them, three or four different options are available for this particular task.

We will present them in roughly chronological order according to their ages. The original FTP proxy for PF is described below in the Section called *FTP through NAT: ftp-proxy*. We then move on to two newer, intermediate solutions developed by Camiel Dobbelaar in the Section called *FTP, PF and routable addresses: ftpsesame, pftpx and ftp-proxy!* before finally moving on to the modern FTP proxy which was introduced in OpenBSD 3.9 in the Section called *ftp-proxy, new style*.

FTP through NAT: ftp-proxy

OpenBSD 3.8 or earlier *equivalents* only

This section is headed for *purely historical* status when the last PF port to other systems has caught up. In November 2005, the old ftp-proxy (`/usr/libexec/ftp-proxy`) was replaced in OpenBSD-current with the new ftp-proxy, which lives in `/usr/sbin`. This is the software which is included in OpenBSD 3.9 onwards and what you will be using on modern PF versions. See the Section called *ftp-proxy, new style* for details.

The old style ftp-proxy which is a part of the base system on systems which offer a PF version based on OpenBSD3.8 or earlier is usually called via the inetd "super server" via an appropriate `/etc/inetd.conf` entry.[1]

The line quoted here specifies that ftp-proxy runs in NAT mode on the loopback interface, `lo0`:

```
127.0.0.1:8021 stream tcp nowait root /usr/libexec/ftp-proxy \
  ftp-proxy -n
```

This line is by default in your `inetd.conf`, commented out with a **#** character at the beginning of the line. To enable your change, you restart inetd.

On FreeBSD, NetBSD and other rcNG based BSDs you do this with the command

```
FreeBSD$ sudo /etc/rc.d/inetd restart
```

or equivalent. Consult man 8 inetd if you are unsure. At this point inetd is running with your new settings loaded.

Now for the actual redirection. Redirection rules and NAT rules fall into the same rule class. These rules may be referenced directly by other rules, and filtering rules may depend on these rules. Logically, rdr and nat rules need to be defined before the filtering rules.

We insert our rdr rule immediately after the nat rule in our `/etc/pf.conf`

```
rdr on $int_if proto tcp from any to any port ftp -> 127.0.0.1 \
```

[1] You may need to enable inetd by adding a **`inetd_enable="YES"`** line to your `rc.conf` and possibly adjust other inetd related configuration settings.

```
        port 8021
```

In addition, the redirected traffic must be allowed to pass. We achive this with

```
pass in on $ext_if inet proto tcp from port ftp-data to ($ext_if) \
    user proxy flags S/SA keep state
```

Save `pf.conf`, then load the new rules with

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

At this point you will probably have users noticing that FTP works before you get around to telling them what you've done.

This example assumes you are using NAT on a gateway with non routable addresses on the inside.

FTP, PF and routable addresses: ftpsesame, pftpx and ftp-proxy!

In cases where the local network uses official, routable address inside the firewall, I must confess I've had trouble making ftp-proxy work properly. When I'd already spent too much time on the problem, I was rather relieved to find a solution to this specific problen offered by a friendly Dutchman called Camiel Dobbelaar in the form of a daemon called ftpsesame.

Local networks using official addresses inside a firewall are apparently rare enough that I'll skip over any further treatment. If you need this and you are running OpenBSD 3.8 or earlier or one of the other PF enabled operating systems, you could do worse than installing ftpsesame.

On FreeBSD, ftpsesame is available through the ports system as `ftp/ftpsesame`. Alternatively you can download ftpsesame from Sentia at http://www.sentia.org/projects/ftpsesame/.

Once installed and running, ftpsesame hooks into your rule set via an anchor, a named sub-ruleset. The documentation consists of a man page with examples which you can more likely than not simply copy and paste.

ftpsesame never made it into the base system, and Camiel went on to write a new solution to the same set of problems.

The new program, at first called pftpx, is available from http://www.sentia.org/downloads/pftpx-0.8.tar.gz and through the FreeBSD ports system as `ftp/pftpx`. pftpx comes with a fairly complete and well written man page to get you started.

A further developed version, suitably renamed as the new ftp-proxy, became a part of the the OpenBSD base system in time for the OpenBSD 3.9. The new program, `/usr/sbin/ftp-proxy`, and how to set it up, is described in the Section called *ftp-proxy, new style* below.

ftp-proxy, new style

For OpenBSD 3.9 and newer

If you are upgrading to OpenBSD 3.9 or newer equivalents or working from a fresh OpenBSD install, this is the ftp-proxy version to use.

Just like its predecessor, the pftpx successor ftp-proxy configuration is mainly a matter of cut and paste from the man page.

If you are upgrading to the new ftp-proxy from an earlier version, you need to remove the ftp-proxy line from your `inetd.conf` file and restart inetd or disable it altogether if your setup does not require a running inetd.

Next, enable ftp-proxy by adding the following line to your `/etc/rc.conf.local` or `/etc/rc.conf`

```
ftpproxy_flags=""
```

You can start the proxy manually by running **`/usr/sbin/ftp-proxy`** if you like.

Moving on to the `pf.conf` file, you need two anchor definitions in the NAT section:

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

Both are needed, even if your setup does not use NAT. If you are migrating from a previous version, your rule set probably contains the appropriate redirection already. If it does not, you add it:

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

Moving on down to the filtering rules, you add an anchor for the proxy to fill in,

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

and finally a pass rule to let the packets pass from the proxy to the rest of the world

```
pass out proto tcp from $proxy to any port 21 keep state
```

where **`$proxy`** expands to the address the proxy daemon is bound to.

This example covers the simple setup with clients who need to contact FTP servers elsewhere. For other variations and more complicated setups, see the ftp-proxy man page.

If you are looking for 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).

Making your network troubleshooting friendly

Making your network troubleshooting friendly is a potentially large subject. At most times, the debugging or troubleshooting friendliness of your TCP/IP network depends on how you treat the Internet protocol which was designed specifically with debugging in mind, the *Internet Control Message Protocol*, or *ICMP* as it is usually abbreviated.

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 is a lot of ICMP traffic which usually just happens in the background while you are surfing the web, reading mail or transferring files. Routers (you are aware that you are building one, right?) 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 referring to ICMP as either ’just evil’, or, if their understanding runs a little deeper, ’a necessary evil’. The reason for this attitude is purely historical. The reason can be found a few years back when it was discovered that several operating systems contained code in their networking stack which could make a machine running one of the affected systems crash and fall over, or in some cases just do really strange things, with a sufficiently large ICMP request.

One of the companies which was hit hard by this was Microsoft, and you can find rather a lot of material on the ’ping of death’ bug by using your favourite search engine. This all happened in the second half of the 1990s, and all modern operating systems, at least the ones we can read, have thoroughly sanitized their network code since then. At least that’s what we are lead to believe.

One of the early workarounds was to simply block either all ICMP traffic or at least ICMP ECHO, which is what ping uses. Now these rule sets have been around for roughly ten years, and the people who put them there are still scared.

Then, do we let it all through?

The obvious question then becomes, if ICMP is such a good and useful thing, should we not let it all through, all the time? The answer is, 'It depends'.

Letting diagnostic traffic pass unconditionally of course makes debugging easier, but it also makes it relatively easy for others to extract information about your network. That means that a rule like

```
pass inet proto icmp from any to any
```

might not be optimal if you want to cloak the internal workings of your network in a bit of mystery. In all fairness it should also be said that you might find some ICMP 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 let all ICMP traffic from your local net through and let probes from elsewhere stop at your gateway:

```
pass inet proto icmp icmp-type $icmp_types from $localnet \
          to any keep state
pass inet proto icmp icmp-type $icmp_types from any to $ext_if \
          keep state
```

Stopping probes at the gateway might be an attractive option anyway, but let us have a look at a few other options which will show you some of PF's flexibility.

Letting ping through

The rule set we have developed so far has one clear disadvantage: common troubleshooting commands such as ping and traceroute will not work. That may not matter too much to your users, and since it was the ping command which scared people into filtering or blocking ICMP traffic in the first place, there are apparently some people who feel we are better off without it. If you are in my perceived target audience, you will be rather fond of having those troubleshooting tools avalable. With a couple of small additions to the rule set, they will be. ping uses ICMP, and in order to keep our rule set tidy, we start by defining another macro:

```
icmp_types = "echoreq"
```

and a rule which uses the definition,

```
pass inet proto icmp all icmp-type $icmp_types keep state
```

You may need more or other types of ICMP packets to go through, and you can then expand **`icmp_types`** to a list of those packet types you want to allow.

Helping traceroute

traceroute is another command which is quite 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 rule below works with the traceroute command on all unixes 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 $ext_if inet proto udp from any to any \
             port 33433 >< 33626 keep state
```

Experience so far indicates that traceroute implementations on other operating systems work roughly the same. Except, of course, Microsoft Windows. On that platform, `TRACERT.EXE` uses ICMP ECHO for this purpose. So if you want to let Windows traceroutes through, you only need the first rule. Unix traceroutes 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.

Under any circumstances, this solution was lifted from 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 very valuable resource whenever you need OpenBSD or PF related information.

Path MTU discovery

The last bit I will 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 can not 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.

The way modern TCP/IP implementations work, they expect to be able to determine the right packet size for a connection through a process which simply puts involves sending packets of varying sizes with the ’Do not fragment’ flag set, expecting an ICMP return packet indicating "type 3, code 4", when the upper limit has been reached. Now you don’t need to dive for the RFCs right away. Type 3 means "destination unreachable", while code 4 is short for "fragmentation needed, but the do not fragment flag is set". So if your connections to networks which may have other MTUs than your own seem sub-optimal, and you do not need to be that specific, you could try changing your list of ICMP types slightly to let the Destination unreachable packets through, too:

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

as we can see, this means we do not need to change the pass rule itself:

```
pass inet proto icmp all icmp-type $icmp_types keep state
```

PF lets you filter on all variations of ICMP types and codes, and if you want to delve into what to pass and not of ICMP traffic, 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[1] .

[1] The main internet RFCs describing ICMP and some related techhiques are RFC792, RFC950, RFC1191, RFC1256, RFC2521, rfc2765, while necessary updates for ICMP for IPv6 are found in RFC1885, RFC2463, RFC2466. These documents are available in a number of places on the net, such as the ietf.org (http://www.ietf.org) and faqs.org (http://www.faqs.org) web sites, and probably also via your package system.
It is quite possible that I will return to ICMP filtering in a future advanced section of the tutorial.

Network hygiene: Blocking, scrubbing and so on

Our gateway does not feel quite complete without a few more items in the configuration which will make it behave a bit more sanely towards hosts on the wide net and our local network.

`block-policy`

`block-policy` is an option which can be set in the **`options`** part of the ruleset, which precedes the redirection and filtering rules. This option determines which feedback, if any, PF will give to hosts which try to create connections which are subsequently blocked. The option has two possible values, **`drop`** which drops blocked packets with no feedback, and **`return`** which returns with status codes such as `Connection refused` or similar.

The correct strategy for block policies has been the subject of rather a lot of discussion. We choose to play nicely and instruct our firewall to issue returns:

```
set block-policy return
```

`scrub`

`scrub` is a keyword which enables network packet normalization, causing fragmented packets to be assembled and removing ambiguity. Enabling **`scrub`** provides a measure of protection against certain kinds of attacks based on incorrect handling of packet fragments. A number of supplementing options are available, but we choose the simplest form which is suitable for most configurations.

```
scrub in all
```

Some services, such as NFS, require some specific fragment handling options. This is extensively documented in the PF user guide and man pages provide all the information you could need.

antispoof

antispoof is a common special case of filtering and blocking. This mechanism protects against activity from spoofed or forged IP addresses, mainly by blocking packets appearing on interfaces and in directions which are logically not possible.

We specify that we want to weed out spoofed traffic coming in from the rest of the world and any spoofed packets which, however unlikely, were to originate in our own network:

```
antispoof for $ext_if
antispoof for $int_if
```

Handling non-routable addresses from elsewhere

Even with a properly configured gateway to handle network address translation for your own network, you may find yourself in the unenviable position of having to compensate for other people's misconfigurations.

One depressingly common class of misconfigurations is the kind which lets traffic with non-routable addresses out to the Internet. Traffic from non-routeable addresses have also played a part in several DOS attack techniques, so it may be worth considering explicitly blocking traffic from non-routeable addresses from entering your network.

One possible solution is the one outlined below, which for good measure also blocks any attempt to initiate contact to non-routable 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 drop in quick on $ext_if from $martians to any
block drop out quick on $ext_if from any to $martians
```

Here, the **martians** macro denotes the RFC 1918 addresses and a few other ranges which are 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.

The specific details of how to implement this kind of protection will vary, among other things according to your specific network configuration. Your network design could for example dictate that you include or exclude other address ranges than these.

This completes our simple NATing firewall for a small local network.

A web server and a mail server on the inside

Time passes, and needs change. Rather frequently, a need to run externally accessible services develops. This quite frequently becomes just a little harder because externally visible addresses are either not available or too expensive, and running several other services on a machine which is primarily a firewall is not a desirable option.

The redirection mechanisms in PF makes it relatively easy to keep servers on the inside. If we assume that we need to run a web server which serves up data in clear text (http) and encrypted (https) and in addition we want a mail server which sends and receives e-mail while letting clients inside and outside the local network use a number of well known submission and retrieval protocols, the following lines may be all that’s needed in addition to the rule set we developed earlier:

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

pass proto tcp from any to $webserver port $webports \
   flags S/SA synproxy state
pass proto tcp from any to $emailserver port $email \
   flags S/SA synproxy state
pass proto tcp from $emailserver to any port smtp \
   flags S/SA synproxy state
```

Notice the flag ’synproxy’ in the new rules. This means that PF will handle the connection setup (three way handshake) on behalf of your server or client before handing the connection over to the application. This provides a certain amount of protection against certain types of attacks.

Rule sets for configurations with DMZ networks isolated behind separate network interfaces and in some cases services running on alternative ports will not necessarily be much different from this one.

Taking care of your own - the inside

Everything I’ve said so far is excellent and correct as long as all you are interested in is getting traffic from hosts outside your local net to reach your servers.

If you want the hosts in your local net to be able to use the services on these machines, you will soon see that the traffic originating in your local network most likely never reaches the external interface. The external interface is where all the redirection and translation happens, and consequently the redirections do not quite work from the inside. The problem is common enough that the PF documentation lists four different solutions to the problem.[1] The options listed in the PF user guide are

- ’Split horizon’ DNS, which means configuring your name service to provide one set of replies for requests originating in the local net and a different one for requests from elsewhere
- proxying using software such as nc (NetCat)
- treating the local net as a special case for redirection and NAT.

  We will be looking into this option below.

- Or simply moving your servers to a separate network, aka a ’DMZ’, with only minor changes to your PF rules.

We need to intercept the network packets originating in the local network and handle those connections correctly, making sure any returning traffic is directed to the communication partner who actually originated the connection.

Returning to our previous example, we achieve this by adding these special case rules:

```
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
```

[1] See Redirection and Reflection (http://openbsd.org/faq/pf/rdr.html#reflect) in the PF user guide.

```
nat on $int_if proto tcp from $localnet to $emailserver \
       port $email -> $int_if
```

It is well worth noting that we do not need to touch the **`pass`** rules at all.

I've had the good fortune to witness via email or IRC the reactions of several network admins at the point when the truth about this five line reconfiguration sank in.

Tables make your life easier

By this time you may be thinking that this gets awfully static and rigid. There will after all be some kinds of data which are relevant to filtering and redirection at a given time, but do not deserve to be put into a configuration file! Quite right, and PF offers mechanisms for handling these situations as well. Tables are one such feature, mainly useful as lists of IP addresses which can be manipulated without needing to reload the entire rule set, and where fast lookups are desirable. Table names are always enclosed in angle brackets, ie < >, like this:

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

here, the network **`192.168.2.0/24`** is part of the table, except the address **`192.168.2.5`**, which is excluded using the ! operator (logical NOT). It is 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
```

which in turn is used to initialize the table in `/etc/pf.conf`:

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

Then, for example, you can change one of our earlier rules to read

```
pass inet proto tcp from <clients> to any port $client_out \
         flags S/SA keep state
```

to manage outgoing traffic from your client computers. With this in hand, you can manipulate the table's contents live, such as

```
$ 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 other reboot unless you arrange to store your changes.

You might opt to maintain the on-disk copy of the table using a cron job which dumps the table content to disk at regular intervals, using a command such as **`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 will be performing frequently, you will sooner or later end up writing shell scripts for tasks such as inserting or removing items or replacing table contents. The only real limitations lie in your own needs and your creativity.[1]

We will be returning to some handy uses of tables shortly, including a few programs which interact with tables in useful ways.

[1] One improvement you could consider is rewriting the **martians** macro from the Section called *Handling non-routable addresses from elsewhere* in the chapter called *Network hygiene: Blocking, scrubbing and so on* as a table

Logging

Up to now we have not mentioned much about logging. To my mind logging and by extension keeping track of what goes on in your network, or at least having the ability to get the information easily is an important part of staying in control of the network. Fortunately PF provides the opportunity to log exactly what you want by adding the **log** keyword to the rules you want logged. You may want to limit the amount of data a bit by specifying one interface where the logging is to be done. You do this by adding

```
set loginterface $ext_if
```

and then editing the rules you want to log, such as

```
pass out log from <clients> to any port $email \
        label client-email keep state
```

This causes the traffic to be logged in a binary format which is really only intended to be used as tcpdump input. Note that **log** here only logs the packet which sets up the connection. If you want to log *all* traffic matching the rule, you use **log (all)** in the rule instead[1] .

The *label* part creates a new set of counters for various statistics for the rule. This can be quite convenient if you are invoicing others for bandwidth use, for example.

It is worth noting that from OpenBSD 4.1, the `pflog` interface is cloneable, which means you can configure as many as you need. At the same time, the **log** syntax for each rule was extended to let you specify on a per rule basis which `pflog` interface to log to, ie

```
pass log (all, to pflog2) inet proto tcp from $mailserver to any port smtp
```

to log outgoing SMTP traffic from the host **$mailserver** to elsewhere, with the log data ending up at the `pflog2` interface.

Taking a peek with tcpdump

Once you have enabled logging in one or more rules, PF logs via the `pflog0`

[1] In PF implementations based on OpenBSD 3.7 and earlier, the keyword for this was **log-all**.

interface, and stores binary log data in the log file `/var/log/pflog`. The log file is useful for a permanent record and for those cases where you want to periodically convert some of the data to other formats. However, if you want to look at your traffic in real time, you can tell tcpdump to look at the `pflog0` log interface instead.

Here is what the output from a couple of log rules can look like on a lazy Thursday afternoon:

```
peter@skapet:~$ sudo tcpdump -n -e -ttt -i pflog0
tcpdump: WARNING: pflog0: no IPv4 address assigned
tcpdump: listening on pflog0, link-type PFLOG
Feb 16 16:43:20.152187 rule 0/(match) block in on ep0: 194.54.59.189.2559 >
194.54.107.19.139: [|tcp] (DF)
Feb 16 16:48:26.073244 rule 27/(match) pass in on ep0: 61.213.167.236 >
194.54.107.19: icmp: echo request
Feb 16 16:49:09.563448 rule 0/(match) block in on ep0: 61.152.249.148.80 >
194.54.107.19.55609: [|tcp]
Feb 16 16:49:14.601022 rule 0/(match) block in on ep0: 194.54.59.189.3056 >
194.54.107.19.139: [|tcp] (DF)
Feb 16 16:53:10.110110 rule 0/(match) block in on ep0: 68.194.177.173 >
194.54.107.19: [|icmp]
Feb 16 16:55:54.818549 rule 27/(match) pass in on ep0: 61.213.167.237 >
194.54.107.19: icmp: echo request
Feb 16 16:57:55.577782 rule 27/(match) pass in on ep0: 202.43.202.16 >
194.54.107.19: icmp: echo request
```

The PF User Guide has a section devoted to logging which contains a number of very useful suggestions. Combined with among other things the tcpdump man pages, you should be able to extract any log data you will find useful.

Other log tools you may want to look into

The logs themselves and the various tcpdump options provide you with valuable tools to gain insight into what happens in your network. Not surprisingly, other tools have been developed to operate on PF log data, collect statistics and do various forms of graphing.

Of special note is Damien Miller's `pfflowd`, which collects PF log data, converts to Cisco *NetFlow*™ for further processing. Damien also develops and maintains see `flowd` for *NetFlow*™ collecting purposes. This Cisco originated data format is supported by a number of different products, and the ability to generate data in this format may be important in certain environments.

One other log data application which is well worth noting is Daniel Hartmeier's `pfstat`, which collects statistics from PF logs and generates graphs from the data. It's a fairly flexible package which takes a lot of the heavy lifting out of presenting log data.

But there are limits (an anecdote)

It might feel tempting at first to put something like this in

```
block log all
```

- just to make sure you don't miss anything.

The PF user guide contains a detailed description of how to make PF log to a human readable text format via syslog, and this does sound rather attractive. I went through the procedure described there when I set up my first PF configuration at work, and the experience sums up rather neatly: Logging is useful, but by all means, be selective. After a little more than an hour the PF text log file had grown to more than a gigabyte, on a machine with less than ten gigabytes of disk space total.

The explanation is simply that even in a rather unexciting Internet backwater, at the far end of an unexceptional ADSL line there's still an incredible amount of uncontrolled Windows traffic such as file sharing and various types of searches trying to get to you. The Windows boxes on the inside probably weren't totally quiet either. At any rate: put some sensible limit on what you log, or make arrangements for sufficient disk space, somewhere.

Keeping an eye on things with pftop

If you are interested in keeping an eye on what passes in to and out of your network, Can Erkin Acar's pftop is a very useful tool. The name is a strong hint at what it does - pftop shows a running snapshot of your traffic in a format which is strongly inspired by top(1):

```
pfTop: Up State 1-21/67, View: default, Order: none, Cache: 10000       19:52:28

PR   DIR SRC                  DEST                  STATE   AGE   EXP  PKTS BYTES
tcp  Out 194.54.103.89:3847   216.193.211.2:25        9:9     28    67    29  3608
tcp  In  207.182.140.5:44870  127.0.0.1:8025          4:4     15 86400    30  1594
tcp  In  207.182.140.5:36469  127.0.0.1:8025        10:10   418    75   810 44675
tcp  In  194.54.107.19:51593  194.54.103.65:22        4:4    146 86395   158 37326
tcp  In  194.54.107.19:64926  194.54.103.65:22        4:4    193 86243   131 21186
tcp  In  194.54.103.76:3010   64.136.25.171:80        9:9    154    59    11  1570
tcp  In  194.54.103.76:3013   64.136.25.171:80        4:4      4 86397     6  1370
tcp  In  194.54.103.66:3847   216.193.211.2:25        9:9     28    67    29  3608
tcp  Out 194.54.103.76:3009   64.136.25.171:80        9:9    214     0     9  1490
tcp  Out 194.54.103.76:3010   64.136.25.171:80        4:4     64 86337     7  1410
udp  Out 194.54.107.18:41423  194.54.96.9:53          2:1     36     0     2   235
udp  In  194.54.107.19:58732  194.54.103.66:53        1:2     36     0     2   219
udp  In  194.54.107.19:54402  194.54.103.66:53        1:2     36     0     2   255
udp  In  194.54.107.19:54681  194.54.103.66:53        1:2     36     0     2   271
```

Your connections can be shown sorted by a number of different criteria, among others by PF rule, volume, age and so on.

This program is not in the base system itself, probably because it is possible to extract equivalent information using various pfctl options. pftop is however available as a package, in ports on OpenBSD and FreeBSD both as `sysutils/pftop`, on NetBSD via pkgsrc as `sysutils/pftop`.

Invisible gateway - bridge

A *bridge* in our context is a machine with two or more network interfaces, located in between the Internet and one or more internal networks, and the network interfaces are not assigned IP addresses. If the machine in question runs OpenBSD or a similarly capable operating system, it is still able to filter and redirect traffic. The advantage of such a setup is that attacking the firewall itself is more difficult. The disadvantage is that all admin tasks must be performed at the firewall's console, unless you configure a network interface which is reachable via a secured network of some kind, or even a serial console.

The exact method for configuring bridges differs in some details between the operating systems. Below is a short recipe for use on OpenBSD, which for good measure blocks all non-Internet protocol traffic. Setting up a bridge with two interfaces:

```
/etc/hostname.ep0

up

/etc/hostname.ep1

up

/etc/bridgename.bridge0

  add ep0 add ep1 blocknonip ep0 blocknonip ep1 up

/etc/pf.conf

ext_if = ep0
int_if  = ep1
interesting-traffic = { ... }
block all
pass quick on $ext_if all
pass log on $int_if from $int_if to any port $interesting-traffic \
     keep state
```

Significantly more complicated setups are possible. Experienced bridgers recommend picking one of the interfaces to perform all filtering and redirection. All packets pass through PF's view twice, making for potentially extremely complicated rules.

In addition, the OpenBSD brconfig command offers its own set of filtering options in addition to other configuration options. The bridge(4) and brconfig(8) man pages offer further information.

FreeBSD uses a slightly different set of commands to configure bridges, while the NetBSD PF implementation supports bridging only with a slightly customized kernel[1] .

[1] See The NetBSD PF documentation (http://www.netbsd.org/Documentation/network/pf.html) for details.

Directing traffic with ALTQ

ALTQ - short for ALTernate Queueing - is a very flexible mechanism for directing network traffic which lived a life of its own before getting integrated into PF. Altq was another one of those things which were integrated into PF because of the additional convenience it offered when integrated.

Altq uses the term *queue* about the main traffic control mechanisms. Queues are defined with a defined amount of bandwidth or a specific part of available bandwidth, where a queue can be assigned subqueues of various types.

To complete the picture, you write filtering rules which assign packets to specified queues or a selection of subqueues where packets pass according to specified criteria.

Queues are created with one of several queue *disciplines*. The default queue discipline without ALTQ is FIFO (first, in first out).

A slightly more interesting discipline is the class based discipline (CBQ), which in practical terms means you define the queue's bandwidth as a set amount of data per second, as a percentage or in units of kilobits, megabits and so on, with an additional priority as an option, or priority based (priq), where you assign priority only.

Priorities can be set at 0 to 7 for cbq queues, 0 to 15 for priq queues, with a higher value assigning a higher priority and preferential treatment. In addtion, the hierarcical queue algoritm "Hierarchical Fair Service Curve" or HFSC is available.

Briefly, a simplified syntax is

```
altq on interface type [options ... ] main_queue { sub_q1, sub_q2 ..}
  queue sub_q1 [ options ... ]
  queue sub_q2 [ options ... ]
[...]
pass [ ... ] queue sub_q1
pass [ ... ] queue sub_q2
```

If you will be using these features in you own rule sets, you should under any circumstances read the `pf.conf` man page and the PF user guide. These documents offer a very detailed and reasonably well laid out explanation of the syntax and options.[1] [2]

ALTQ - prioritizing by traffic type

Our first real example is lifted from Daniel Hartmeier's web. Like quite a few of us, Daniel is on an asymmetric connection, and naturally he wanted to get better bandwidth utilization.

One symptom in particular seemed to indicate that there was room for improvement. Incoming traffic (downloads) apparently slowed down outgoing traffic.

Analyzing the data indicated that the ACK packets for each data packet transferred caused a disproportionately large slowdown, possibly due to the FIFO (First In, First Out) queue discipline in effect on the outgoing traffic.

A testable hypothesis formed - if the tiny, practically dataless ACK packets were able to slip inbetween the larger data packets, this would lead to a more efficient use of available bandwidth. The means were two queues with different priorities. The relevant parts of the rule set follows:

```
ext_if="kue0"

altq on $ext_if priq bandwidth 100Kb queue { q_pri, q_def }
queue q_pri priority 7
queue q_def priority 1 priq(default)

pass out on $ext_if proto tcp from $ext_if to any flags S/SA \
        keep state queue (q_def, q_pri)

pass in  on $ext_if proto tcp from any to $ext_if flags S/SA \
        keep state queue (q_def, q_pri)
```

The result was indeed better performance.

[1] On FreeBSD, ALTQ requires the ALTQ and queue discipline options for the disciplines you want to use to be compiled into the running kernel. Refer to the PF chapter (http://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/firewalls-pf.html) of the FreeBSD Handbook for further information.

[2] At the time of writing, ALTQ isn in the process of getting integrated in the NetBSD 4.0 PF implementation. For earlier NetBSD versions, Peter Postma maintains a patch to enable PF/ALTQ functions. Up to date information on this, including how to get the ALTQ patch via pkgsrc, is available from Peter Postmas PF on NetBSD pages, http://nedbsd.nl/~ppostma/pf/ and The NetBSD PF documentation (http://www.netbsd.org/Documentation/network/pf.html).

So why does this work?

So why does this work?[3] The reason lies in how the ALTQ code treats subqueues with different priorities. Once a connection is assigned to the main queue, ALTQ inspects each packet's type of service (ToS) field. ACK packets have the ToS Delay bit set to 'low', which indicates that the sender wanted the speediest delivery possible.

When ALTQ sees a low delay packet and queues of differing priorities are available, it will assign the packet to the higher priority queue. This means that the ACK packets skip ahead of the lower priority queue and are delivered more quickly, which in turn means that data packets are serviced more quickly, too.

Daniel's article is available from his web site at
http://www.benzedrine.cx/ackpri.html

ALTQ - allocation by percentage

We move on to another example, which I for all practical purposes swiped from the Swedish site unix.se. The queues are set up on the external interface. This is probably the more common approach, since the limitations on bandwidth are usually more severe on the external interface. In principle, however, allocating queues and running traffic shaping can be done on any network interface. Here, the setup includes a cbq queue for a total bandwidth of 640KB with six sub queues.

```
altq on $ext_if cbq bandwidth 640Kb queue { def, ftp, udp, http, \
        ssh, icmp }
queue def bandwidth 18% cbq(default borrow red)
queue ftp bandwidth 10% cbq(borrow red)
queue udp bandwidth 30% cbq(borrow red)
queue http bandwidth 20% cbq(borrow red)
queue ssh bandwidth 20% cbq(borrow red) { ssh_interactive, ssh_bulk }
queue ssh_interactive priority 7
queue ssh_bulk priority 0
queue icmp bandwidth 2% cbq
```

We see the subqueue def with 18 percent of the bandwidth is designated as the default queue, that is any traffic not explicitly assigned to some other

[3] Earlier versions of this tutorial left the explanation pretty much as an exercise to the reader.

queue ends up here. The borrow and red keywords mean that the queue may 'borrow' bandwidth from its parent queue, while the system attempts to avoid congestion by applying the RED (Random Early Detection) algorithm. The other queues follow more or less the same pattern, up to the subqueue ssh, which itself has two subqueues with separate priorities.

In the ssh queue, again we see a variation of the ACK priority via subqueues scheme: Bulk SSH transfers, typically SCP file transfers, get transmitted with a ToS indicating normal delay, while interactive SSH traffic has the low delay bit set and skips ahead of the bulk transfers.

This scheme probably also helps the speed of SCP file transfers, since the SCP ACK packets will be assigned to the higher priority subqueue.

Finally, the pass rules which show which traffic gets assigned to the queues, and their criteria:

```
pass log quick on $ext_if proto tcp from any to any port 22 flags S/SA \
    keep state queue (ssh_bulk, ssh_interactive)
pass in quick on $ext_if proto tcp from any to any port 20 flags S/SA  \
    keep state queue ftp
pass in quick on $ext_if proto tcp from any to any port 80 flags S/SA \
    keep state queue http
pass out on $ext_if proto udp all keep state queue udp
pass out on $ext_if proto icmp all keep state queue icmp
```

We can reasonably assume that this allocation meets the site's needs.

The full description can be found at the Unix.se site as http://unix.se/Brandv%E4gg_med_OpenBSD

ALTQ - handling unwanted traffic

Our last altq example is one which surfaced around the time of one of the many spam or virus storms we've seen during the last few years. It's fairly common knowledge that the machines causing these bursts of email are practically all Windows machines. PF has a fairly reliable operating system fingerprinting mechanism which detects the operating system at the other end of a network connection. One OpenBSD user got sufficiently tired of all this meaningless traffic, and posted some selected bits of his `pf.conf` on his blog:

```
altq on $ext_if cbq queue { q_default q_web q_mail }
```

```
   queue q_default cbq(default)
   queue q_web (...)

    ## all mail limited to 1Mb/sec
    queue q_mail bandwidth 1Mb { q_mail_windows }
    ## windows mail limited to 56Kb/sec
    queue q_mail_windows bandwidth 56Kb

    pass in quick proto tcp from any os "Windows" to $ext_if port 25 \
            keep state queue q_mail_windows
    pass in quick proto tcp from any to $ext_if port 25 label "smtp" \
            keep state queue q_mail
```

```
" I can’t believe I didn’t see this earlier. Oh, how sweet. ...
  Already a huge difference in my load. Bwa ha ha. "
```

Randal L. Schwartz, 29. january 2004,
http://use.perl.org/~merlyn/journal/17094

Here all email traffic is assigned one megabit worth of bandwidth, while email traffic originating at Windows hosts get to share a subqueue of 56 kbit total. No wonder the total load went down and the blog post ends with what must be an evil chuckle.

I must confess this is something I’ve wanted very much to do myself, but I’ve never dared. A few too many of our customers have for their own reasons chosen to run their mail service on Windows, and we do like to receive most of their mail. In a little while, we’ll have a look at a different PF approach which may have achieved much of the same effect.

CARP and pfsync

CARP and pfsync were two of the main new items in OpenBSD 3.5. CARP is short for Common Address Redundancy Protocol. The protocol was developed as a non patent encumbered alternative to VRRP (Virtual Router Redundancy Protocol, RFC 2281, RFC 3768), which was quite far along the track to becoming an IETF sanctioned standard, even though possible patent issued has not been resolved. The patents involved are held by Cisco, IBM and Nokia.

Both protocols are intended to ensure redundancy for essential network features, with automatic failover.

CARP is based on setting up a group of machines as one ’master’ and one of more redundant ’slaves’, all of which are equipped to handle a common IP address. If the master goes down, one of the slaves will inherit the IP address, and if the synchronization has been properly handled, active connections will be handed over. The handover may be authenticated using cryptographic keys.

One of the main purposes of CARP is to ensure that the network will keep functioning as usual even when a firewall or other service goes down due to errors or planned maintenance activities such as upgrades.

In the case of PF firewalls, pfsync can handle the synchronization. pfsync is a type of virtual network interface specially designed to synchronize state information between PF firewalls. pfsync interfaces are assigned to physical interfaces with ifconfig. On networks where uptime requirements are strict enough to dictate automatic failover, the number of simultaneous network connections is likely to be large enough that it will make sense to assign the pfsync network its own physical network.

I hope to explore these exciting and advanced features in a future advanced section of the tutorial. At this point, the best pfsync and CARP references are the OpenBSD FAQ, the man pages and Ryan McBride’s overview article at http://www.countersiege.com/doc/pfsync-carp/.

Wireless networks made simple

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

A little IEEE 802.11 background

Moving to wireless networks provides an opportunity to view security at various level in the networking stack from a new perspective. We look briefly at two of the basic IEEE 802.11 security mechanisms below.[1]

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

WEP (Wired Equivalent Privacy)

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

WEP is a link level encryption scheme which is considered a pretty primitive homebrew among cryptography professionals. It was no great surprise that WEP encryption was reverse-engineered and cracked within a few months after the first products were released. Even though you can download tools for free to descramble WEP encoded traffic in a matter of minutes, for a variety of reasons it is still widely supported and used. You

[1] For a more complete overview of issues surrounding security in wireless networks, see eg Professor Kjell Jørgen Hole's articles and slides at www.kjhole.com (http://www.kjhole.com/Standards/WiFi/WiFiDownloads.html). For fresh developments in the WiFI field, the sites wifinetnews.com (http://wifinetnews.com/archives/cat_security.html) and The Unofficial 802.11 Security Web Page (http://www.drizzle.com/~aboba/IEEE/) come higly recommended.

should consider network traffic protected only by WEP to be only 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 (WiFi Protected Access)

It dawned fairly quickly on the 802.11 designers that their Wired Equivalent Privacy system was not quite what it was cracked up to be, and they came up with a revised and slightly more comprehensive solution which was dubbed *WiFi Protected Access*, or *WPA*.

WPA looks better than WEP, at least on paper, but the specification is complicated enough that it is not nearly as widely supported or implemented as its designers intended. In addition WPA has also attracted its share of criticism over design issues and bugs. Combined with the familiar issues of access to documentation and hardware, free software support varies. If your project specification includes WPA, look carefully at your operating system and driver documentation.

Setting up a simple wireless network

The first part is to make sure you have a supported card and check your dmesg output to see that the driver loads and initializes the card properly[2]

[2] Wireless network support in OpenBSD and BSDs in general is getting better all the time, but this does not mean that getting all the bits you need is necessarily easy. A brief history of my home network goes like this: I started out buying two CNet CWP-854 cards, which should be supported in OpenBSD 3.7 via the new ral driver. The one I put in the brand new Dell machine running a non-free operating system worked right out of the box. My gateway, which had been running without incident since the 3.3 days however, was a little more problematic. The card did get recognized and configured, but once the Dell tried to get an IP address, the gateway went down with a kernel panic. The gory details are available as OpenBSD PR number 4217. I have promised to test the card again with a new snapshot - as soon as I can locate the card again. From the Dell we could see an amazing number of networks, almost all unsecured, but that's another story entirely.
I decided i wanted to try `ath` cards, and bought a D-Link DWL-G520, which I then managed to misplace while moving house. Next, I bought a DWL-G520+, thinking that the plus sign must mean it's better. Unfortunately, the plus meant a whole different chipset was used, the TI ACX111, which comes with a low price tag but with no documentation accessible to free software developers. Fortunately the store let me return the card for a refund with no trouble at all. At this point, I was getting rather frustrated, and went all across town to a shop which had several DWL-AG520 cards in stock. It was a bit more expensive than the others, but it did

With a successfully configured card you should see someting like

```
ath0 at pci1 dev 4 function 0 "Atheros AR5212" rev 0x01: irq 11
ath0: AR5212 5.6 phy 4.1 rf5111 1.7 rf2111 2.3, ETSI1W, address
00:0d:88:c8:a7:c4
```

Next, you need to configure the interface for TCP/IP. On OpenBSD, this means an `/etc/hostname.ath0` roughly like this:

```
up media autoselect mediaopt hostap mode 11b chan 6 nwid unwiredbsd \
 nwkey 0x1deadbeef9
inet 10.168.103.1
```

Note that the configuration is divided over two lines. The first line generates an ifconfig command which sets up the interface with the correct parameters for the physical wireless network, the second command, which gets executed only after the first one completes, sets the IP address. Note that we set the channel explicitly, and we enable a weak WEP encryption by setting the **`nwkey`** parameter.

On FreeBSD you would need to put those lines in your `/etc/start_if.ath0`, and substitute your interface name for **`ath0`** if required

Then you most likely want to set up dhcpd to serve addresses and other relevant network information to clients. Your clients would need an `/etc/hostname.ath0` configuration of

```
up media autoselect mode 11b chan 6 nwid unwiredbsd nwkey 0x1deadbeef9
dhcp
```

and again on FreeBSD, you would need to put those lines in your `/etc/start_if.ath0`, and substitute your interface name for **`ath0`** here if required.

Assuming your gateway does NAT, you will want to set up NAT for the wireless network as well, by making some small changes to your

work right away. A couple of weeks later the G520 turned up, and of course that worked too. My laptop (which at the time ran FreeBSD) came with a Realtek 8180 wireless mini-PCI card built in, but for some reason I could not get it to work. I ended up buying DWL-AG650 cardbus card, which works flawlessly with the `ath` driver. In general, my advice is, if you shop online, keep the man pages available in another tab or window, and if you go to a physical store, make sure to tell the clerks you will be using a BSD, and if you're not sure about the card they are trying to sell you, see if you can borrow a machine to browse the online man pages. Telling the clerks up front could end up making it easier to get a refund if the part does not work, and telling them the card did work is good advocacy.
It is possibly worth noting that the `acx` driver, introduced in OpenBSD 4.0, has brought reverse engineered support for ACX1nn based cards to the BSDs.

`/etc/pf.conf`:

```
air_if = "ath0"
```

and

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

You will need a similar near duplicate line for your ftp-proxy config, and include **`$air_if`** in your pass rules.

That's all there is to it. This configuration gives you a functional BSD access point, with at least token security via WEP encryption.

An open, yet tightly guarded wireless network with authpf

As always, there are other ways to configure the security of your wireless network than the one we have just seen. What little protection WEP encryption offers, security professionals tend to agree is barely enough to signal to an attacker that you do not intend to let all and sundry use your network resources.

A different approach appeared one day in my mail as a message from my friend Vegard Engen, who told me he had been setting up authpf. authpf is a user shell which lets you 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 apply to the IP address which 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 connection 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.

Vegard's annotated config follows below. His wireless network is configured without WEP encryption, preferring to handle the security side of things via PF and authpf:

Start with creating an empty `/etc/authpf/authpf.conf`. It needs to be there for authpf to work, but doesn't actually need any content.

The other relevant bits of `/etc/pf.conf` follow. First, interface macros:

```
int_if="sis1"
ext_if="sis0"
wi_if = "wi0"
```

The use of this address will become apparent later:

```
auth_web="192.168.27.20"
```

The traditional authpf table

```
table <authpf_users> persist
```

We could put the NAT part in `authpf.rules`, but keeping it in the main `pf.conf` doesn't hurt:

```
nat on $ext_if from $wi_if:network to any -> ($ext_if)
```

Redirects to let traffic reach servers on the internal net. These could be put in `authpf.rules` too, but since they do not actually provide access without pass rules, keeping them here won't hurt anything.

```
rdr on $wi_if proto tcp from any to $myaddr port $tcp_in -> $server
rdr on $wi_if proto udp from any to $myaddr port $udp_in -> $server
```

The next redirect sends all web traffic from non authenticated users to port 80 on **`$auth_web`**. In Vegard's setup, this is a web server which displays contact info for people who stumble onto the wireless net. In a commercial setting, this would be where you would put something which could handle credit cards and create users.

```
rdr on $wi_if proto tcp from ! <authpf_users> to any \
 port 80 -> $auth_web
```

To activate nat, binat or redirects in authpf

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

On to the filtering rules, we start with a sensible default

```
block all
```

Other global, user independent rules would go here. Next for the authpf anchor, we make sure non-authenticated users connecting to the wireless interface get redirected to **`$auth_web`**

```
anchor "authpf/*" in on wi0

pass in on $wi_if inet proto tcp from any to $auth_web \
 port 80 keep state
```

There are three things we want anyway on the wireless interface: Name service (DNS), DHCP and SSH in to the gateway. Three rules do the trick

```
pass in on $wi_if inet proto udp from any port 53 keep state

pass in on $wi_if inet proto udp from any to $wi_if port 67

pass in on $wi_if inet proto tcp from any to $wi_if \
 port 22 keep state
```

Next up, the we define the rules which get loaded for all users who log in with their shell set to `/usr/sbin/authpf`. These rules go in `/etc/authpf/authpf.rules`,

```
int_if = "sis1"
ext_if = "sis0"
wi_if = "wi0"
server = "192.168.27.15"
myaddr = "213.187.n.m"

# Services which live on the internal network
# and need to be accessible
tcp_services = "{ 22, 25, 53, 80, 110, 113, 995 }"
udp_services = "{ 53 }"
tcp_in = " { 22, 25, 53, 80, 993, 2317, pop3}"
udp_in = "{ 53 }"

# Pass traffic to elsewhere, that is the outside world
pass in on $wi_if inet from <authpf_users> to ! $int_if:network \
   keep state

# Let authenticated users use services on
# the internal network.

pass in on $wi_if inet proto tcp from <authpf_users> to $server \
  port $tcp_in keep state
pass in on $wi_if inet proto udp from <authpf_users> to $server \
  port $udp_in keep state

# Also pass to external address. This means you can access
# internal services on external addesses.

pass in on $wi_if inet proto tcp from <authpf_users> to $myaddr \
    port $tcp_in keep state
pass in on $wi_if inet proto udp from <authpf_users> to $myaddr \
    port $udp_in keep state
```

At this point we have an open net where anybody can connect and get an IP address from DHCP. All HTTP requests from non-authenticated users get redirected to port 80 on 192.168.27.20, which is a server on the internal net where all requests are answered with the same page, which displays contact info in case you want to be registered and be allowed to use the net.

You are allowed to ssh in to the gateway. Users with valid user IDs and passwords get rule sets with appropriate pass rules loaded for their assigned IP address.

We can fine tune this even more by making user specific rules in `/etc/authpf/users/$user/authpf.rules`. Per user rules can use the $user_ip macro for the user's IP address. For example, if I want to give myself unlimited access, create the following `/etc/authpf/users/vegard/authpf.rules`:

```
wi_if="wi0"
pass in on $wi_if from $user_ip to any keep state
```

Turning away the brutes

If you run a Secure Shell login service anywhere which is accessible from the Internet, I'm sure you've seen things like these in your authentication logs:

```
Sep 26 03:12:34 skapet sshd[25771]: Failed password for root from
200.72.41.31 port 40992 ssh2
Sep 26 03:12:34 skapet sshd[5279]: Failed password for root from
200.72.41.31 port 40992 ssh2
Sep 26 03:12:35 skapet sshd[5279]: Received disconnect from
200.72.41.31: 11: Bye Bye
Sep 26 03:12:44 skapet sshd[29635]: Invalid user admin from
200.72.41.31
Sep 26 03:12:44 skapet sshd[24703]: input_userauth_request:
invalid user admin
Sep 26 03:12:44 skapet sshd[24703]: Failed password for invalid user
admin from 200.72.41.31 port 41484 ssh2
Sep 26 03:12:44 skapet sshd[29635]: Failed password for invalid user
admin from 200.72.41.31 port 41484 ssh2
Sep 26 03:12:45 skapet sshd[24703]: Connection closed by 200.72.41.31
Sep 26 03:13:10 skapet sshd[11459]: Failed password for root from
200.72.41.31 port 43344 ssh2
Sep 26 03:13:10 skapet sshd[7635]: Failed password for root from
200.72.41.31 port 43344 ssh2
Sep 26 03:13:10 skapet sshd[11459]: Received disconnect from
200.72.41.31: 11: Bye Bye
Sep 26 03:13:15 skapet sshd[31357]: Invalid user admin from 200.72.41.31
Sep 26 03:13:15 skapet sshd[10543]: input_userauth_request: invalid
user admin
Sep 26 03:13:15 skapet sshd[10543]: Failed password for invalid user
admin from 200.72.41.31 port 43811 ssh2
Sep 26 03:13:15 skapet sshd[31357]: Failed password for invalid user
admin from 200.72.41.31 port 43811 ssh2
Sep 26 03:13:15 skapet sshd[10543]: Received disconnect from
200.72.41.31: 11: Bye Bye
Sep 26 03:13:25 skapet sshd[6526]: Connection closed by 200.72.41.31
```

It gets repetetive after that. This is what a brute force attack looks like. Essentially somebody, or more likely, a cracked computer somewhere, is trying by brute force to find a combination of user name and password which will let them into your system.

The simplest response would be to write a `pf.conf` rule which blocks all access. This leads to another class of problems, including what you do in order to let people with legitimate business on your system access it anyway. You might consider moving the service to some other port, but then again, the ones flooding you on port 22 would probably be able to scan their way to port 22222 for a repeat performance.

Since OpenBSD 3.7[1], PF has offered a slightly more elegant solution. You can write your pass rules so they maintain certain limits on what connecting hosts can do. For good measure, you can banish violators to a table of addresses which you deny some or all access. You can even choose to drop all existing connections from machines which overreach your limits, if you like. Here's how it's done:

Now first set up the table. In your tables section, add

```
table <bruteforce> persist
```

Then somewhere fairly early in your rule set you set up to block from the bruteforcers

```
block quick from <bruteforce>
```

And finally, your pass rule.

```
pass inet proto tcp from any to $localnet port $tcp_services \
        flags S/SA keep state \
(max-src-conn 100, max-src-conn-rate 15/5, \
         overload <bruteforce> flush global)
```

This is rather similar to what we've seen before, isn't it? In fact, the first part is identical to the one we constructed earlier. The part in brackets is the new stuff which will ease your network load even further.

max-src-conn is the number of simultaneous connections you allow from one host. In this example, I've set it at 100, in your setup you may want a slightly higher or lower value.

max-src-conn-rate is the rate of new connections allowed from any single host, here 15 connections per 5 seconds. Again, you are the one to judge what suits your setup.

overload `<bruteforce>` means that any host which exceeds these limits gets its address added to the table `bruteforce`. Our rule set blocks all traffic from addresses in the bruteforce table.

[1] Introduced to FreeBSD in version 6.0

finally, *flush global* says that when a host reaches the limit, that host's connections will be terminated (flushed). The global part says that for good measure, this applies to connections which match other pass rules too.

The effect is dramatic. My bruteforcers more often than not end up with `"Fatal: timeout before authentication"` messages, which is exactly what we want.

Once again, please keep in mind that this example rule is intended mainly as an illustration. It is not unlikely that your network's needs are better served by rather different rules or combinations of rules.

If, for example, you want to allow a generous number of connections in general, but would like to be a little more tight fisted when it comes to ssh, you could supplement the rule above with something like the one below, early on in your rule set:

```
pass quick proto { tcp, udp } from any to any port ssh \
        flags S/SA keep state \
        (max-src-conn 15, max-src-conn-rate 5/3, \
        overload <bruteforce> flush global)
```

You should be able to find the set of parameters which is just right for your situation by reading the relevant man pages and the PF User Guide (http://www.openbsd.org/faq/pf/), and perhaps a bit of experimentation.

You may not need to `block` all of your `overloaders`

It is probably worth noting at this point that the *overload* mechanism is a general technique which does not have to apply exclusively to the *ssh* service, and it is not necessarily always optimal to block all traffic from offenders entirely.

You could for example use an overload rule to protect a mail service or a web service, and you could use the overload table in a rule to assign offenders to a queue with a minimal bandwidth allocation (see the Section called *ALTQ - handling unwanted traffic* in the chapter called *Directing traffic with ALTQ*) or, in the web case, to redirect to a specific web page (much like in the authpf example in the chapter called *An open, yet tightly guarded wireless network with authpf*).

expiretable tidies your tables

At this point, we have tables which will be filled by our **`overload`** rules, and since we could reasonably expect our gateways to have months of uptime, the tables will grow incrementally, taking up more memory as time goes by.

You could also find that an IP address you blocked last week due to a brute force attack was in fact a dynamically assigned one, which is now assigned to a different ISP customer who has a legitimate reason to try communicating with hosts in your network.

Situations like these were what caused Henrik Gustafsson to write expiretable, which removes table entries which have not been accessed for a specified period of time.

One useful example is to use the expiretable program as a way of removing outdated <**`bruteforce`**> table entries.

You could for example let expiretable remove <**`bruteforce`**> table entries older than 24 hours by adding an entry containing the following to your `/etc/rc.local` file:

```
/usr/local/sbin/expiretable -v -d -t 24h bruteforce
```

expiretable was recently added to the ports tree on FreeBSD and OpenBSD[2].

If expiretable is not available via your package system, you can download it from Henrik's site at http://expiretable.fnord.se/

expiring table entries with pfctl

In OpenBSD 4.1, pfctl acquired the ability to expire table entries not referenced in a specified number of seconds. For example, the command

```
# pfctl -t bruteforce -T expire 86400
```

will remove `<bruteforce>` table entries which have not been referenced for 86400 seconds.

[2] as `security/expiretable` and `sysutils/expiretable`, respectively

Giving spammers a hard time

At this point we've covered quite some ground, and I'm more than happy to present something really useful: PF as a means to make spammers' lives harder. Based on our recent exposure to PF rulesets, understanding the following `/etc/pf.conf` parts should be straighforward:

```
table <spamd> persist
table <spamd-white> persist
rdr pass on $ext_if inet proto tcp from <spamd> to \
         { $ext_if, $localnet } port smtp -> 127.0.0.1 port 8025
rdr pass on $ext_if inet proto tcp from !<spamd-white> to \
         { $ext_if, $localnet } port smtp -> 127.0.0.1 port 8025
```

We have two tables, for now it's sufficient to note their names and the fact that these names have a special meaning in this context. SMTP traffic from the addresses in the first table above plus the ones which are not in the other table are redirected to a daemon listening at port 8025.

The application which uses these tables, spamd, is a fake SMTP daemon, designed to waste spammers' time and keep their traffic off our net. That's what lives at port 8025, and the last part of our session here will be centered around how to make good use of that software.

Remember, you are not alone: blacklisting

The main point underlying the spamd design is the fact that spammers send a large number of messages, and the probability that you are the first person receiving a particular message is incredibly small. In addition, spam is mainly sent via a few spammer friendly networks and a large number of hijacked machines. Both the individual messages and the machines will be reported to blacklists fairly quickly, and this is the data which eventually ends up in the first table in our example.

List of black and grey, and the sticky tarpit

What spamd does to SMTP connections from addresses in the blacklist is to present its banner and immediately switch to a mode where it answers SMTP traffic 1 byte at the time. This technique, which is intended to waste as much time as possible on the sending end while costing the receiver pretty much nothing, is called *tarpitting*. The specific implementation with 1 byte SMTP replies is often referred to as *stuttering*.

spamd also supports *greylisting*, which works by rejecting messages from unknown hosts temporarily with 45n codes, letting messages from hosts which try again within a reasonable time through. Traffic from well behaved hosts, that is, senders which are set up to behave within the limits set up in the relevant RFCs[1], will be let through.

Greylisting as a technique was presented in a 2003 paper by Evan Harris[2], and a number of implementations followed over the next few months. OpenBSD's spamd aquired its ability to greylist in version OpenBSD 3.5, which was released in May 2004. Starting with OpenBSD 4.1, spamd by default runs in greylisting mode.

The most amazing thing about greylisting, apart from its simplicity, is that it still works. Spammers and malware writers have been very slow to adapt. We will see a few examples later.

Setting up spamd

With the necessary rules in place in your `pf.conf`, configuring spamd is fairly straightforward[3]. You simply edit your `spamd.conf` (traditionally stored in the `/etc` directory, but on OpenBSD 4.1 and newer the file has migrated to `/etc/mail`), according to your own needs. The file itself offers quite a bit of explanation, and the man page offers additional information, but we will recap the essentials here.

[1] The relevant RFCs are mainly RFC1123 and RFC2821. If you choose to join us greylisting pedants, you will need to read these, if only for proper RFC-style background information. Remember, temporary rejection is in fact an SMTP fault tolerance feature.

[2] The original Harris paper and a number of other useful articles and resources can be found at the greylisting.org (http://www.greylisting.org/) web site.

[3] Note that on FreeBSD, spamd is a port, `mail/spamd/`. If you are running PF on FreeBSD 5.n or newer, you need to install the port, follow the directions given by the port's messages and return here.
In particular, to use spamd's greylisting features, you need to have a file descriptor file system (see **`man 5 fdescfs`**) mounted at `/dev/fd/`. You do this by adding the following line to your `/etc/fstab`:

```
    fdescfs /dev/fd fdescfs rw 0 0
```

and making sure the `fdescfs` code is in your kernel, either compiled in or by loading the module via the appropriate kldload command.

One of the first lines without a **#** comment sign at the start contains the block which defines the **all** list, which specifies the lists you actually use:

```
all:\
:becks:whitelist:
```

Here you add all black lists you want to use, separated by colons (:). If you want to use whitelists to subtract addresses from your blacklist, you add the name of the whitelist immediately after the name of each blacklist, ie **:blacklist:whitelist:**.

Next up is a blacklist definition:

```
becks:\
        :black:\
        :msg="SPAM. Your address %A has sent spam within the last 24 hours":\
        :method=http:\
        :file=www.openbsd.org/spamd/traplist.gz
```

Following the name, the first data field specifies the list type, in this case **black**. The *msg* field contains the message to display to blacklisted senders during the SMTP dialogue. The *method* field specifies how spamd-setup fetches the list data, here **http**. The other options are fetching via **ftp**, from a **file** in a mounted file system or via **exec** of an external program. Finally the *file* field specifies the name of the file spamd expects to receive.

The definition of a whitelist follows much the same pattern:

```
whitelist:\
        :white:\
        :method=file:\
        :file=/etc/mail/whitelist.txt
```

but omits the message parameter since a message is not needed.

Choose your data sources with care

Enabling the suggested blacklists in the default as distributed `spamd.conf` could lead to blacklisting of quite large blocks of the Internet, including several countries such as Korea. I work in a company which actually does the odd bit of business with Koreans, and consequently I needed to edit out that particular entry from our configuration. You are the judge of which data sources to use, and using other lists than the default ones is possible.

Put the lines for spamd and the startup parameters you want in your `/etc/rc.conf` or `/etc/rc.conf.local`, for example

```
spamd_flags="-v -G 2:4:864" # for normal use: "" and see spamd-setup(8)
spamd_grey=YES              # use spamd greylisting if YES
```

Once again, on OpenBSD 4.1 onwards, the **`spamd_grey`** variable is superfluous. If you want spamd to run in pure blacklist mode without greylisting, you use the **`spamd_black`** variable to turn off greylisting and enable blacklisting mode.

Note for that you can fine tune several of the greylisting related parameters via spamd command line parameters. Check the spamd man page to see what the parameters mean.

When you are done with editing the setup, you start spamd with the options you want, and complete the configuration using spamd-setup. Finally, you create a cron job which calls spamd-setup to update the tables at reasonable intervals.

Once the tables are filled, you can view table contents using pfctl or other applications. If you want to change or delete entries, you are advised to use the spamdb utility instead of pfctl table commands. More about that later.

Note that the example above uses rdr rules which are also pass rules. If your rdr rules do not include a 'pass' part, you need to set up pass rules to let traffic through to your redirection. You also need to set up rules to let legitimate email through. If you are already running an email service on your network, you can probably go on using your old SMTP pass rules.

Some early highlights of our spamd experience

What is spamd like in practical use? We started using spamd in earnest in early December of 2004, after running spamassassin and clamav as parts of the exim delivery process for a while. Our exim is configured to tag and deliver messages with a spamassassin score in the interval from 5 to 9.99 points, while discarding messages with 10 points or more along with malware carrying messages. As the autumn progressed, spamassassin's success rate had been steadily declining, letting ever more spam through.

When we put spamd into production, the total number of messages handled and the number of messages handled by spamassassin decreased drastically. The number of spam messages which make it through untagged is now stabilized at roughly five a day, based on a reporting population of a handful of users.

If you start spamd with the -v command line option for verbose logging, the logs start including a few more items of information in addition to the IP addresses. With verbose logging, a typical log excerpt looks like this:

```
Oct  2 19:55:05 delilah spamd[26905]: (GREY) 83.23.213.115:
<gilbert@keyholes.net> -> <wkitp98zpu.fsf@datadok.no>
Oct  2 19:55:05 delilah spamd[26905]: 83.23.213.115: disconnected after
0 seconds.
Oct  2 19:55:05 delilah spamd[26905]: 83.23.213.115: connected (2/1)
Oct  2 19:55:06 delilah spamd[26905]: (GREY) 83.23.213.115:
<gilbert@keyholes.net> -> <wkitp98zpu.fsf@datadok.no>
Oct  2 19:55:06 delilah spamd[26905]: 83.23.213.115: disconnected after
1 seconds.
Oct  2 19:57:07 delilah spamd[26905]: (BLACK) 65.210.185.131:
 <bounce-3C7E40A4B3@branch15.summer-bargainz.com> -> <adm@dataped.no>
Oct  2 19:58:50 delilah spamd[26905]: 65.210.185.131: From: Auto
lnsurance Savings <noreply@branch15.summer-bargainz.com>
Oct  2 19:58:50 delilah spamd[26905]: 65.210.185.131: Subject: Start
SAVlNG M0NEY on Auto lnsurance
Oct  2 19:58:50 delilah spamd[26905]: 65.210.185.131: To: adm@dataped.no
Oct  2 20:00:05 delilah spamd[26905]: 65.210.185.131: disconnected after
404 seconds. lists: spews1
Oct  2 20:03:48 delilah spamd[26905]: 222.240.6.118: connected (1/0)
Oct  2 20:03:48 delilah spamd[26905]: 222.240.6.118: disconnected after
0 seconds.
Oct  2 20:06:51 delilah spamd[26905]: 24.71.110.10: connected (1/1),
lists: spews1
Oct  2 20:07:00 delilah spamd[26905]: 221.196.37.249: connected (2/1)
Oct  2 20:07:00 delilah spamd[26905]: 221.196.37.249: disconnected after
0 seconds.
Oct  2 20:07:12 delilah spamd[26905]: 24.71.110.10: disconnected after
21 seconds. lists: spews1
```

The first three lines say that a machine connects, as the second active connection, with one connection from a blacklisted host. The `(GREY)` and `(BLACK)` before the addresses indicate greylisting or blacklisting status, respectively. After 404 seconds (or 6 minutes, 44 seconds), the blacklisted host gives up without completing the delivery. The following lines may be the first ever contact from a machine, which is then greylisted.[4]

[4] Note the rather curious local part (user name) of the address in the message which the greylisted machine tries to deliver here. There is more to this story.

At the time this tutorial was originally written, our preliminary conclusion was that spamd was quite efficient in stopping spam. Unfortunately, along the way we encountered some false positives. Indications are that the false positives came from a few too broadly defined entries in the spews2 (spews level 2) list. For now we have stopped using this list as a blacklist, without a noticeable increase in the spam volume.

Now for what used to be the the climax of my spamd experience. One log entry stood out for a long time:

```
Dec 11 23:57:24 delilah spamd[32048]: 69.6.40.26: connected (1/1),
lists: spamhaus spews1 spews2
Dec 12 00:30:08 delilah spamd[32048]: 69.6.40.26: disconnected
after 1964 seconds. lists: spamhaus spews1 spews2
```

This entry concerns a sender at wholesalebandwidth.com. This particular machine made 13 attempts at delivery during the period from December 9th to December 12th, 2004. The last attempt lasted 32 minutes, 44 seconds, without completing the delivery.

I update the tutorial now and again, and recently I found a few more entries which exceeded this:

```
peter@delilah:~$ grep disconnected /var/log/spamd | awk '{print $9}' \
| sort -rn | uniq -c | head
   1 42673
   1 36099
   1 14714
   1 10170
   1 5495
   1 3025
   1 2193
   1 1964
   1 1872
   1 1718
```

The first, at 42673 seconds, which is almost twelve hours,

```
Dec 21 14:22:44 delilah spamd[29949]: 85.152.224.147: connected (5/2)
Dec 21 14:22:46 delilah spamd[29949]: 85.152.224.147: connected (6/2)
Dec 21 14:22:47 delilah spamd[29949]: 85.152.224.147: disconnected
after 3 seconds.
Dec 22 02:13:59 delilah spamd[29949]: 85.152.224.147: disconnected
after 42673 seconds.
```

concerns a host which is apparently in a Spanish telecoms operator's network. The machine was probably infected by an extremely naive spam sending worm, which just took a long time waiting for the rest of the SMTP dialogue. The next entries, at 10 hours, 1 minute, 39 seconds, 4 hours, 5 minutes and 14 seconds and 2 hours, 49 minutes and 30 seconds respectively, seem to have behaved in much the same way.

Beating'em up some more: spamdb and greytrapping

Behind the scenes, rarely mentioned and barely documented are two of spamd's helpers, the spamdb database tool and the spamlogd whitelist updater, which both perform essential functions for the greylisting feature. Of the two spamlogd works quietly in the background, while spamdb has been developed to offer some interesting features.

Restart spamd to enable greylisting

If you followed all steps in the tutorial exactly up to this point, spamlogd has been started automatically already. However, if your initial spamd configuration did not include greylisting, spamlogd may not have been started, and you may experience strange symptoms, such as your greylists and whitelist not getting updated properly.

Under normal circumstances, you should not have to start spamlogd by hand. Restarting spamd after you have enabled greylisting ensures spamlogd is loaded and available too.

spamdb is the administrator's main interface to managing the black, grey and white lists via the contents of the `/var/db/spamdb` database.

Early versions of spamdb simply offered options to add whitelist entries to the database or update existing ones (**`spamdb -a nn.mm.nn.mm`**) and to delete whitelist entries (**`spamdb -d nn.mm.nn.mm`**) to compensate for shortcomings in either the blacklists used or the effects of the greylisting algorithms.

By the time the development cycle for OpenBSD 3.8 started during the first half of 2005, spamd users and developers had accumulated significant amounts of data and experience on spammer behaviour and spammer reactions to countermeasures.

We already know that spam senders rarely use a fully compliant SMTP implementation to send their messages. That's why greylisting works. Also, as we noted earlier, not only do spammers send large numbers of messages, they rarely check that the addresses they feed to their hijacked machines are actually deliverable. Combine these facts, and you see that if a greylisted machine tries to send a message to an invalid address in your domain, there is a significant probability that the message is a spam, or for that matter, malware.

Enter greytrapping

Consequently, spamd had to learn *greytrapping*. Greytrapping as implemented in spamd puts offenders in a temporary blacklist, dubbed `spamd-greytrap`, for 24 hours. Twenty-four hours is short enough to not cause serious disruption of legitimate traffic, since real SMTP implementations will keep trying to deliver for a few days at least. Experience from large scale implementations of the technique shows that it rarely if ever produces false positives[5] . Machines which continue spamming after 24 hours will make it back to the tarpit soon enough.

Your own traplist

To set up your own traplist, you use spamdb's **-T** option. In my case, the strange address I mentioned earlier[6] was a natural candidate for inclusion:

```
peter@delilah:~$ spamdb -T -a "<wkitp98zpu.fsf@datadok.no>"
```

Sure enough, the spammers thought this was just as usable as almost two years ago:

[5] One prime example is Bob Beck's "ghosts of usenet postings past" based traplist, which rarely contains less than 20,000+ entries. The number of hosts varies widely and has been as high as roughly 90,000. At the time of writing (mid February, 2007), the list typically contained around 65,000 entries. While still officially in testing, the list was made publicly available on January 30th, 2006. The list has to my knowledge yet to produce any false positives and is available from http://www.openbsd.org/spamd/traplist.gz for your `spamd.conf`.

[6] That address is completely bogus. It is probably based on a GNUS message-ID, which in turn was probably lifted from a news spool or some unfortunate malware victim's mailbox.

```
Nov  6 09:50:25 delilah spamd[23576]: 210.214.12.57: connected (1/0)
Nov  6 09:50:32 delilah spamd[23576]: 210.214.12.57: connected (2/0)
Nov  6 09:50:40 delilah spamd[23576]: (GREY) 210.214.12.57:
<gilbert@keyholes.net> -> <wkitp98zpu.fsf@datadok.no>
Nov  6 09:50:40 delilah spamd[23576]: 210.214.12.57: disconnected
after 15 seconds.
Nov  6 09:50:42 delilah spamd[23576]: 210.214.12.57: connected (2/0)
Nov  6 09:50:45 delilah spamd[23576]: (GREY) 210.214.12.57:
<bounce-3C7E40A4B3@branch15.summer-bargainz.com> ->
<adm@dataped.no>
Nov  6 09:50:45 delilah spamd[23576]: 210.214.12.57: disconnected
after 13 seconds.
Nov  6 09:50:50 delilah spamd[23576]: 210.214.12.57: connected (2/0)
Nov  6 09:51:00 delilah spamd[23576]: (GREY) 210.214.12.57:
<gilbert@keyholes.net> -> <wkitp98zpu.fsf@datadok.no>
Nov  6 09:51:00 delilah spamd[23576]: 210.214.12.57: disconnected
after 18 seconds.
Nov  6 09:51:02 delilah spamd[23576]: 210.214.12.57: connected (2/0)
Nov  6 09:51:02 delilah spamd[23576]: 210.214.12.57: disconnected
after 12 seconds.
Nov  6 09:51:02 delilah spamd[23576]: 210.214.12.57: connected (2/0)
Nov  6 09:51:18 delilah spamd[23576]: (GREY) 210.214.12.57:
<gilbert@keyholes.net> -> <wkitp98zpu.fsf@datadok.no>
Nov  6 09:51:18 delilah spamd[23576]: 210.214.12.57: disconnected
after 16 seconds.
Nov  6 09:51:18 delilah spamd[23576]: (GREY) 210.214.12.57:
<bounce-3C7E40A4B3@branch15.summer-bargainz.com> ->
<adm@dataped.no>
Nov  6 09:51:18 delilah spamd[23576]: 210.214.12.57: disconnected
after 16 seconds.
Nov  6 09:51:20 delilah spamd[23576]: 210.214.12.57: connected (1/1),
lists: spamd-greytrap
Nov  6 09:51:23 delilah spamd[23576]: 210.214.12.57: connected (2/2),
lists: spamd-greytrap
Nov  6 09:55:33 delilah spamd[23576]: (BLACK) 210.214.12.57:
<gilbert@keyholes.net> -> <wkitp98zpu.fsf@datadok.no>
Nov  6 09:55:34 delilah spamd[23576]: (BLACK) 210.214.12.57:
<bounce-3C7E40A4B3@branch15.summer-bargainz.com> ->
<adm@dataped.no>
```

This log fragment shows how the spammer's machine is greylisted at first contact, and then clumsily tries to deliver messages to my greytrap address, only to end up in the `spamd-greytrap` blacklist after a few minutes. By now we all know what it will be doing for the next twenty-odd hours.

Deleting, handling trapped entries

spamdb offers a few more options you should be aware of. The `-T` option combined with `-d` lets you delete traplist mail address entries, while the `-t` (lowercase) option comined with `-a` or `-d` lets you add or delete trapped IP address entries from the database.

Exporting your list of currently trapped addresses can be as simple as putting together a simple one-liner with spamdb, grep and a little imagination.

The downside: some people really do not get it

We have already learned that the main reason why greylisting works is that any standards compliant mail setup is required to retry delivery after some “reasonable” amount of time. However as Murphy will be all too happy to tell you, life is not always that simple.

For one thing, the first email message sent from any site which has not contacted you for as long as the greylister keeps its data around will be delayed for some random amount of time which depends mainly on the sender’s retry interval. There are some circumstances where avoiding even a minimal delay is desirable. If you for example have some infrequent customers who always demand your immediate and urgent attention to their business when they do contact you, an initial delivery delay of what could be several hours may not be optimal.

In addition, you are bound to encounter misconfigured mail servers which either do not retry at all or retry too quickly, perhaps stopping delivery retries after a few attempts. As luck would have it, in your case one of these is likely to be at an important customer’s site, run by an incompentent who will not listen to reason or possiby a site owned and operated by your boss’ boyfriend.

Finally, there are some sites which are large enough to have several outgoing SMTP servers, and not play well with greylisting since they are not guaranteed to retry delivery of any given message from the same IP address as the last delivery attempt for that message. Even though those sites can sincerely claim to comply with the retry requirements, since the RFCs do no state that the new delivery attempts *have to* come from the same IP address, it’s fairly obvious that this is one of the few remaining downsides of greylisting.

If you need to compensate for such things in your setup, it is fairly easy to do. One useful approach is to define a table for a local whitelist, to be fed from a file in case of reboots:

```
table <localwhite> file "/etc/mail/whitelist.txt"
```

To make sure SMTP traffic from the addresses in the table is not fed to spamd, you add a **`no rdr`** rule at the top of your redirection block:

```
no rdr proto tcp from <localwhite> to $mailservers port smtp
```

Once you have these changes added to your rule set, you enter the addresses you need to protect from redirection into the `whitelist.txt` file, then reload your rule set using **`pfctl -f`**. You can then use all the expected table tricks on the <**`localwhite`**> table, including replacing its content after editing the `whitelist.txt` file. See the chapter called *Tables make your life easier* or **`man pfctl`** for a few pointers.

Conclusions from our spamd experience

Summing up, selectively used, blacklists combined with spamd are powerful, precise and efficient spam fighting tools. The load on the spamd machine is minimal. On the other hand, spamd will never perform better than its weakest data source, which means you will need to monitor your logs and use whitelisting when neccessary.

It is also perfectly feasible to run spamd in a pure greylisting mode, with no blacklists. In fact some users report that a purely greylisting spamd configuration is not significantly less effective than blacklising configurations as spam fighting tools[7].

For our main blacklist, we ended up using Bob Beck's traplist[8], which is generated using "the ghosts of USENET postings past", that is, the spamd's greytrapping feature and addresses which are not expected to receive legitimate mail. What makes this list stand out is that Bob set up the

[7] One such report is Steve Williams' October 20th, 2006 message to the OpenBSD-misc mailing list (http://marc.info/?l=openbsd-misc&m=116136841831550&w=2), where he reports that a pure greylisting configuration immediately rid the company he worked for of approximately 95% of their spam load.

[8] The list is hosted at http://www.openbsd.org/spamd/traplist.gz.

system to remove addresses automatically after 24 hours. This means that you get an extremely low number of false positives.

Once you're happy with your setup, you could try introducing local greytrapping. This is likely to catch a few more undesirables, and of course it's good clean fun.

PF - Haiku

Finally, an indication of the level of feeling inspired by PF in its users is in order. On the PF mailing list, a message with the subject of "Things pf can't do?" appeared in May 2004. The message had been written by someone who did not have a lot of firewalls experience, and who consequently found it hard to get the setup he or she wanted.

This, of course, lead to some discussion, with several participants saying that if PF was hard on a newbie, the alternatives were certainly not a bit better. The thread ended in the following haiku of praise from Jason Dixon, which is given intact as it came, along with Jason's comments:

```
Compared to working with iptables, PF is like this haiku:

A breath of fresh air,
floating on white rose petals,
eating strawberries.

Now I'm getting carried away:

Hartmeier codes now,
Henning knows not why it fails,
fails only for n00b.

Tables load my lists,
tarpit for the asshole spammer,
death to his mail store.

CARP due to Cisco,
redundant blessed packets,
licensed free for me.
```

Jason Dixon, on the PF email list, May 20th, 2004
(http://marc.info/?l=openbsd-pf&m=108507584013046&w=2)

References

OpenBSDs web http://www.openbsd.org/

OpenBSDs FAQ, http://www.openbsd.org/faq/index.html

PF User Guide http://www.openbsd.org/faq/pf/index.html

Daniel Hartmeier's PF pages, http://www.benzedrine.cx/pf.html

Daniel Hartmeier: Design and Performance of the OpenBSD Stateful Packet Filter (pf), http://www.benzedrine.cx/pf-paper.html (presented at Usenix 2002)

Nate Underwood: HOWTO: Transparent Packet Filtering with OpenBSD, http://ezine.daemonnews.org/200207/transpfobsd.html

Randal L. Schwartz: Monitoring Net Traffic with OpenBSD's Packet Filter, http://www.samag.com/documents/s=9053/sam0403j/0403j.htm

Unix.se: Brandvägg med OpenBSD, http://unix.se/Brandv%E4gg_med_OpenBSD

Randal L. Schwartz: Blog for Thu, Jan 29, 2004, http://use.perl.org/~merlyn/journal/17094

RFC 1631, "The IP Network Address Translator (NAT)", May 1994 http://www.ietf.org/rfc/rfc1631.txt?number=1631

RFC 1918, "Address Allocation for Private Internets", February 1996 http://www.ietf.org/rfc/rfc1918.txt?number=1918

The FreeBSD PF home page, http://pf4freebsd.love2party.net/

Peter Postma's PF on NetBSD pages, http://nedbsd.nl/~ppostma/pf/

Marcus Ranum: The Six Dumbest Ideas in Computer Security (http://www.ranum.com/security/computer_security/editorials/dumb/index.html), September 1, 2005

Kjell Jørgen Hole WiFi courseware, http://www.kjhole.com/Standards/WiFi/WiFiDownloads.html, also see wifinetnews.com (http://wifinetnews.com/archives/cat_security.html); also The Unofficial 802.11 Security Web Page (http://www.drizzle.com/~aboba/IEEE/) comes higly recommended.

Greylisting.org greylisting.org (http://www.greylisting.org/) is the home of all things greylisting, with links to numerous articles and other useful information.

Evan Harris: The Next Step in the Spam Control War: Greylisting (http://greylisting.org/articles/whitepaper.shtml) (the original greylisting paper)

Peter N. M. Hansteen: The silent network: Denying the spam and malware chatter using free tools (http://home.nuug.no/~peter/malware-talk/silent-network.pdf) - paper presented at BSDCan 2007 which puts spamd into a slightly wider spam and malware fighting context along with some data on spammer behavior

Where to find the tutorial on the web

Work in progress edition:

Several formats http://home.nuug.no/~peter/pf/

This is where updated versions will appear more or less in the natural course of tinkering, in between events. Please let me know if you want to be told of future updates.

If you enjoyed this: Buy OpenBSD CDs and other items, donate!

If you have enjoyed this tutorial or found it useful, please go to the OpenBSD.org Orders page (http://www.openbsd.org/orders.html) and buy CD sets, or for that matter, support further development work by the OpenBSD project via a donation (http://www.openbsd.org/donations.html).

If you are in a position to donate hardware, make sure you check OpenBSD developers' hardware wishlist at http://www.openbsd.org/want.html and get in contact with the right people in the project to arrange the details.

If you've taken this in at a conference, there might even be an OpenBSD booth nearby where you can buy CDs, T-shirts or other items.

Remember, even free software takes real work and real money to develop and maintain.