IP Filter Based Firewalls HOWTO

Brendan Conoboy <synk@swcp.com>
Erik Fichtner <emf@obfuscation.org>

Wed Dec 11 12:42:58 EST 2002

Abstract: This document is intended to introduce a new user to the IP Filter firewalling package and, at the same time, teach the user some basic fundamentals of good firewall design.

1. Introduction

IP Filter is a great little firewall package. It does just about everything other free firewalls (ipfwadm, ipchains, ipfw) do, but it's also portable and does neat stuff the others don't. This document is intended to make some cohesive sense of the sparse documentation presently available for ipfilter. Futhermore, this document will also serve as documentation for syntatically compatible packet filters (notably *pf*) and, wherever applicable, will point out any important differences; however this document is specifically targeted at the rules language and syntax, and is not intended to be platform specific. Some prior familiarity with packet filtering will be useful, however too much familiarity may make this document a waste of your time. For greater understanding of firewalls, the authors recommend reading *Building Internet Firewalls*, Chapman & Zwicky, O'Reilly and Associates; and *TCP/IP Illustrated, Volume 1*, Stevens, Addison-Wesley.

1.1. Disclaimer

The authors of this document are not responsible for any damages incurred due to actions taken based on this document. This document is meant as an introduction to building a firewall based on IP-Filter. If you do not feel comfortable taking responsibility for your own actions, you should stop reading this document and hire a qualified security professional to install your firewall for you.

1.2. Copyright

1.3. Where to obtain the important pieces

The official IPF homepage is at: *<http://coombs.anu.edu.au/~avalon/ip-filter.html>*

The BSD licensed compatible packet filter, pf, is at: *<http://www.benzedrine.cx/pf.html>*

The most up-to-date version of this document can be found at: *<http://www.obfuscation.org/ipf/>*

2. Basic Firewalling

This section is designed to familiarize you with ipfilter's syntax, and firewall theory in general. The features discussed here are features you'll find in any good firewall package. This section will give you a good foundation to make reading and understanding the advanced section very easy. It must be emphasized that this section alone is not enough to build a good firewall, and that the advanced section really is required reading for anybody who wants to build an effective security system.

2.1. Config File Dynamics, Order and Precedence

IPF (IP Filter) has a config file (as opposed to say, running some command again and again for each new rule). The config file drips with Unix: There's one rule per line, the "#" mark denotes a comment, and you can have a rule and a comment on the same line. Extraneous whitespace is allowed, and is encouraged to keep the rules readable.

2.2. Basic Rule Processing

The rules are processed from top to bottom, each one appended after another. This quite simply means that if the entirety of your config file is:

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

The computer sees it as:

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

Which is to say that when a packet comes in, the first thing IPF applies is:

```
block in all
```

Should IPF deem it necessary to move on to the next rule, it would then apply the second rule:

```
pass  in all
```

At this point, you might want to ask yourself "would IPF move on to the second rule?" If you're familiar with ipfwadm or ipfw, you probably won't ask yourself this. Shortly after, you will become bewildered at the weird way packets are always getting denied or passed when they shouldn't. Many packet filters stop comparing packets to rulesets the moment the first match is made; IPF is not one of them.

Unlike the other packet filters, IPF keeps a flag on whether or not it's going to pass the packet. Unless you interrupt the flow, IPF will go through the entire ruleset, making its decision on whether or not to pass or drop the packet based on the last matching rule. The scene: IP Filter's on duty. It's been been scheduled a slice of CPU time. It has a checkpoint clipboard that reads:

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

A packet comes in the interface and it's time to go to work. It takes a look at the packet, it takes a look at the first rule:

```
block in all
```

"So far I think I will block this packet" says IPF. It takes a look at the second rule:

```
pass  in all
```

"So far I think I will pass this packet" says IPF. It takes a look at a third rule. There is no third rule, so it goes with what its last motivation was, to pass the packet onward.

It's a good time to point out that even if the ruleset had been

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

that the packet would still have gone through. There is no cumulative effect. The last matching rule always takes precedence.

2.3. Controlling Rule Processing

If you have experience with other packet filters, you may find this layout to be confusing, and you may be speculating that there are problems with portability with other filters and speed of rule matching. Imagine if you had 100 rules and most of the applicable ones were the first 10. There would be a terrible overhead for every packet coming in to go through 100 rules every time. Fortunately, there is a simple keyword you can add to any rule that makes it take action at that match. That keyword is `quick`.

Here's a modified copy of the original ruleset using the `quick` keyword:

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

In this case, IPF looks at the first rule:

```
block in quick all
```

The packet matches and the search is over. The packet is expunged without a peep. There are no notices, no logs, no memorial service. Cake will not be served. So what about the next rule?

```
pass  in       all
```

This rule is never encountered. It could just as easily not be in the config file at all. The sweeping match of `all` and the terminal keyword `quick` from the previous rule make certain that no rules are followed afterward.

Having half a config file laid to waste is rarely a desirable state. On the other hand, IPF is here to block packets and as configured, it's doing a very good job. Nonetheless, IPF is also here to let some packets through, so a change to the ruleset to make this possible is called for.

2.4. Basic filtering by IP address

IPF will match packets on many criteria. The one that we most commonly think of is the IP address. There are some blocks of address space from which we should never get traffic. One such block is from the unroutable networks, 192.168.0.0/16 (/16 is the CIDR notation for a netmask. You may be more familiar with the dotted decimal format, 255.255.0.0. IPF accepts both). If you wanted to block 192.168.0.0/16, this is one way to do it:

```
block in quick from 192.168.0.0/16 to any
pass  in       all
```

Now we have a less stringent ruleset that actually does something for us. Let's imagine a packet comes in from 1.2.3.4. The first rule is applied:

```
block in quick from 192.168.0.0/16 to any
```

The packet is from 1.2.3.4, not 192.168.*.*, so there is no match. The second rule is applied:

```
pass  in       all
```

The packet from 1.2.3.4 is definitely a part of `all`, so the packet is sent to whatever it's destination happened to be.

On the other hand, suppose we have a packet that comes in from 192.168.1.2. The first rule is applied:

```
block in quick from 192.168.0.0/16 to any
```

There's a match, the packet is dropped, and that's the end. Again, it doesn't move to the second rule because the first rule matches and contains the `quick` keyword.

At this point you can build a fairly extensive set of definitive addresses which are passed or blocked. Since we've already started blocking private address space from entering our firewall, let's take care of the rest of it:

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

The first three address blocks are some of the private IP space.†

2.5. Controlling Your Interfaces

It seems very frequent that companies have internal networks before they want a link to the outside world. In fact, it's probably reasonable to say that's the main reason people consider firewalls in the first place. The machine that bridges the outside world to the inside world and vice versa is the router. What separates the router from any other machine is simple: It has more than one interface.

Every packet you receive comes from a network interface; every packet you transmit goes out a network interface. Say your machine has 3 interfaces, `lo0` (loopback), `xl0` (3com ethernet), and `tun0` (FreeBSD's generic tunnel interface that PPP uses), but you don't want packets coming in on the `tun0` interface?

```
block in quick on tun0 all
pass  in               all
```

In this case, the `on` keyword means that that data is coming in on the named interface. If a packet comes `in on tun0`, the first rule will block it. If a packet comes `in on lo0` or `in on xl0`, the first rule will not match, the second rule will, the packet will be passed.

2.6. Using IP Address and Interface Together

It's an odd state of affairs when one decides it best to have the `tun0` interface up, but not allow any data to be received from it. The more criteria the firewall matches against, the tighter (or looser) the firewall can become. Maybe you want data from `tun0`, but not from 192.168.0.0/16? This is the start of a powerful firewall.

```
block in quick on tun0 from 192.168.0.0/16 to any
pass  in       all
```

---

† See rfc1918 at <*http://www.faqs.org/rfcs/rfc1918.html*> and <*http://www.ietf.org/internet-drafts/draft-manning-dsua-06.txt*>

Compare this to our previous rule:

```
block in quick from 192.168.0.0/16 to any
pass  in       all
```

The old way, all traffic from 192.168.0.0/16, regardless of interface, was completely blocked. The new way, using `on tun0` means that it's only blocked if it comes in on the `tun0` interface. If a packet arrived on the `xl0` interface from 192.168.0.0/16, it would be passed.

At this point you can build a fairly extensive set of definitive addresses which are passed or blocked. Since we've already started blocking private address space from entering `tun0`, let's take care of the rest of it:

```
block in quick on tun0 from 192.168.0.0/16 to any
block in quick on tun0 from 172.16.0.0/12 to any
block in quick on tun0 from 10.0.0.0/8 to any
block in quick on tun0 from 127.0.0.0/8 to any
block in quick on tun0 from 0.0.0.0/8 to any
block in quick on tun0 from 169.254.0.0/16 to any
block in quick on tun0 from 192.0.2.0/24 to any
block in quick on tun0 from 204.152.64.0/23 to any
block in quick on tun0 from 224.0.0.0/3 to any
pass  in       all
```

You've already seen the first three `blocks`, but not the rest. The fourth is a largely wasted class-A network used for loopback. Much software communicates with itself on 127.0.0.1 so blocking it from an external source is a good idea. The fifth, 0.0.0.0/8, should never be seen on the internet. Most IP stacks treat "0.0.0.0/32" as the default gateway, and the rest of the 0.*.*.* network gets handled strangely by various systems as a byproduct of how routing decisions are made. You should treat 0.0.0.0/8 just like 127.0.0.0/8. 169.254.0.0/16 has been assigned by the IANA for use in auto-configuration when systems have not yet been able to obtain an IP address via DHCP or the like. Most notably, Microsoft Windows will use addresses in this range if they are set to DHCP and cannot find a DHCP server. 192.0.2.0/24 has also been reserved for use as an example IP netblock for documentation authors. We specifically do not use this range as it would cause confusion when we tell you to block it, and thus all our examples come from 20.20.20.0/24. 204.152.64.0/23 is an odd netblock reserved by Sun Microsystems for private cluster interconnects, and blocking this is up to your own judgement. Lastly, 224.0.0.0/3 wipes out the "Class D and E" networks which is used mostly for multicast traffic, although further definition of "Class E" space can be found in RFC 1166.

There's a very important principle in packet filtering which has only been alluded to with the private network blocking and that is this: When you know there's certain types of data that only comes from certain places, you setup the system to only allow that kind of data from those places. In the case of the unroutable addresses, you know that nothing from 10.0.0.0/8 should be arriving on `tun0` because you have no way to reply to it. It's an illegitimate packet. The same goes for the other unroutables as well as 127.0.0.0/8.

Many pieces of software do all their authentication based upon the packet's originating IP address. When you have an internal network, say 20.20.20.0/24, you know that the only traffic for that internal network is going to come off the local ethernet. Should a packet from 20.20.20.0/24 arrive over a PPP dialup, it's perfectly reasonable to drop it on the floor, or put it in a dark room for interrogation. It should by no means be allowed to get to its final destination. You can accomplish this particularly easily with what you already know of IPF. The new ruleset would be:

```
block in quick on tun0 from 192.168.0.0/16 to any
block in quick on tun0 from 172.16.0.0/12 to any
block in quick on tun0 from 10.0.0.0/8 to any
block in quick on tun0 from 127.0.0.0/8 to any
block in quick on tun0 from 0.0.0.0/8 to any
block in quick on tun0 from 169.254.0.0/16 to any
block in quick on tun0 from 192.0.2.0/24 to any
block in quick on tun0 from 204.152.64.0/23 to any
block in quick on tun0 from 224.0.0.0/3 to any
block in quick on tun0 from 20.20.20.0/24 to any
pass  in       all
```

2.7. Bi-Directional Filtering; The "out" Keyword

Up until now, we've been passing or blocking inbound traffic. To clarify, inbound traffic is all traffic that enters the firewall on any interface. Conversely, outbound traffic is all traffic that leaves on any interface (whether locally generated or simply passing through). This means that all packets coming in are not only filtered as they enter the firewall, they're also filtered as they exit. Thusfar there's been an implied `pass out all` that may or may not be desirable.† Just as you may pass and block incoming traffic, you may do the same with outgoing traffic.

Now that we know there's a way to filter outbound packets just like inbound, it's up to us to find a conceivable use for such a thing. One possible use of this idea is to keep spoofed packets from exiting your own network. Instead of passing any traffic out the router, you could instead limit permitted traffic to packets originating at 20.20.20.0/24. You might do it like this:

```
pass  out quick on tun0 from 20.20.20.0/24 to any
block out quick on tun0 from any to any
```

If a packet comes from 20.20.20.1/32, it gets sent out by the first rule. If a packet comes from 1.2.3.4/32 it gets blocked by the second.

You can also make similar rules for the unroutable addresses. If some machine tries to route a packet through IPF with a destination in 192.168.0.0/16, why not drop it? The worst that can happen is that you'll spare yourself some bandwidth:

```
block out quick on tun0 from any to 192.168.0.0/16
block out quick on tun0 from any to 172.16.0.0/12
block out quick on tun0 from any to 10.0.0.0/8
block out quick on tun0 from any to 0.0.0.0/8
block out quick on tun0 from any to 127.0.0.0/8
block out quick on tun0 from any to 169.254.0.0/16
block out quick on tun0 from any to 192.0.2.0/24
block out quick on tun0 from any to 204.152.64.0/23
block out quick on tun0 from any to 224.0.0.0/3
block out quick on tun0 from !20.20.20.0/24 to any
```

In the narrowest viewpoint, this doesn't enhance your security. It enhances everybody else's security, and that's a nice thing to do. As another viewpoint, one might suppose that because nobody can send spoofed packets from your site, that your site has less value as a relay for crackers, and as such is less of a target.

You'll likely find a number of uses for blocking outbound packets. One thing to always keep in mind is that in and out directions are in reference to your firewall, never any other machine.

2.8. Logging What Happens; The "log" Keyword

Up to this point, all blocked and passed packets have been silently blocked and silently passed. Usually you want to know if you're being attacked rather than wonder if that firewall is really buying you any added benefits. While I wouldn't want to log every passed packet, and in some cases every blocked packet, I would want to know about the blocked packets from 20.20.20.0/24. To do this, we add the `log` keyword:

```
block in     quick on tun0 from 192.168.0.0/16 to any
block in     quick on tun0 from 172.16.0.0/12 to any
block in     quick on tun0 from 10.0.0.0/8 to any
block in     quick on tun0 from 127.0.0.0/8 to any
block in     quick on tun0 from 0.0.0.0/8 to any
block in     quick on tun0 from 169.254.0.0/16 to any
block in     quick on tun0 from 192.0.2.0/24 to any
block in     quick on tun0 from 204.152.64.0/23 to any
block in     quick on tun0 from 224.0.0.0/3 to any
block in log quick on tun0 from 20.20.20.0/24 to any
pass  in       all
```

So far, our firewall is pretty good at blocking packets coming to it from suspect places, but there's still more to be done. For one thing, we're accepting packets destined anywhere. One thing we ought to do is make sure packets to 20.20.20.0/32 and 20.20.20.255/32 get dropped on the floor. To do otherwise opens the internal network for a smurf attack. These two lines would prevent our hypothetical network from being used as a smurf relay:

† This can, of course, be changed by using -DIPFILTER_DEFAULT_BLOCK when compiling ipfilter on your system.

```
block in log quick on tun0 from any to 20.20.20.0/32
block in log quick on tun0 from any to 20.20.20.255/32
```

This brings our total ruleset to look something like this:

```
block in     quick on tun0 from 192.168.0.0/16 to any
block in     quick on tun0 from 172.16.0.0/12 to any
block in     quick on tun0 from 10.0.0.0/8 to any
block in     quick on tun0 from 127.0.0.0/8 to any
block in     quick on tun0 from 0.0.0.0/8 to any
block in     quick on tun0 from 169.254.0.0/16 to any
block in     quick on tun0 from 192.0.2.0/24 to any
block in     quick on tun0 from 204.152.64.0/23 to any
block in     quick on tun0 from 224.0.0.0/3 to any
block in log quick on tun0 from 20.20.20.0/24 to any
block in log quick on tun0 from any to 20.20.20.0/32
block in log quick on tun0 from any to 20.20.20.255/32
pass  in       all
```

2.9. Complete Bi-Directional Filtering By Interface

So far we have only presented fragments of a complete ruleset. When you're actually creating your ruleset, you should setup rules for every direction and every interface. The default state of ipfilter is to pass packets. It is as though there were an invisible rule at the beginning which states `pass in all` and `pass out all`. Rather than rely on some default behaviour, make everything as specific as possible, interface by interface, until every base is covered.

First we'll start with the `lo0` interface, which wants to run wild and free. Since these are programs talking to others on the local system, go ahead and keep it unrestricted:

```
pass out quick on lo0
pass in  quick on lo0
```

Next, there's the `xl0` interface. Later on we'll begin placing restrictions on the `xl0` interface, but to start with, we'll act as though everything on our local network is trustworthy and give it much the same treatment as `lo0`:

```
pass out quick on xl0
pass in  quick on xl0
```

Finally, there's the `tun0` interface, which we've been half-filtering with up until now:

```
block out quick on tun0 from any to 192.168.0.0/16
block out quick on tun0 from any to 172.16.0.0/12
block out quick on tun0 from any to 127.0.0.0/8
block out quick on tun0 from any to 10.0.0.0/8
block out quick on tun0 from any to 0.0.0.0/8
block out quick on tun0 from any to 169.254.0.0/16
block out quick on tun0 from any to 192.0.2.0/24
block out quick on tun0 from any to 204.152.64.0/23
block out quick on tun0 from any to 224.0.0.0/3
pass  out quick on tun0 from 20.20.20.0/24 to any
block out quick on tun0 from any to any

block in     quick on tun0 from 192.168.0.0/16 to any
block in     quick on tun0 from 172.16.0.0/12 to any
block in     quick on tun0 from 10.0.0.0/8 to any
block in     quick on tun0 from 127.0.0.0/8 to any
block in     quick on tun0 from 0.0.0.0/8 to any
block in     quick on tun0 from 169.254.0.0/16 to any
block in     quick on tun0 from 192.0.2.0/24 to any
block in     quick on tun0 from 204.152.64.0/23 to any
block in     quick on tun0 from 224.0.0.0/3 to any
block in log quick on tun0 from 20.20.20.0/24 to any
block in log quick on tun0 from any to 20.20.20.0/32
block in log quick on tun0 from any to 20.20.20.255/32
pass  in     all
```

This is a pretty significant amount of filtering already, protecting 20.20.20.0/24 from being spoofed or being used for spoofing. Future examples will continue to show one-sideness, but keep in mind that it's for brevity's sake, and when setting up your own ruleset, adding rules for every direction and every interface is necessary.

2.10. Controlling Specific Protocols; The "proto" Keyword

Denial of Service attacks are as rampant as buffer overflow exploits. Many denial of service attacks rely on glitches in the OS's TCP/IP stack. Frequently, this has come in the form of ICMP packets. Why not block them entirely?

```
block in log quick on tun0 proto icmp from any to any
```

Now any ICMP traffic coming in from `tun0` will be logged and discarded.

2.11. Filtering ICMP with the "icmp-type" Keyword; Merging Rulesets

Of course, dropping all ICMP isn't really an ideal situation. Why not drop all ICMP? Well, because it's useful to have partially enabled. So maybe you want to keep some types of ICMP traffic and drop other kinds. If you want ping and traceroute to work, you need to let in ICMP types 0 and 11. Strictly speaking, this might not be a good idea, but if you need to weigh security against convenience, IPF lets you do it.

```
pass in quick on tun0 proto icmp from any to 20.20.20.0/24 icmp-type 0
pass in quick on tun0 proto icmp from any to 20.20.20.0/24 icmp-type 11
```

Remember that ruleset order is important. Since we're doing everything `quick` we must have our `pass`es before our `block`s, so we really want the last three rules in this order:

```
pass  in     quick on tun0 proto icmp from any to 20.20.20.0/24 icmp-type 0
pass  in     quick on tun0 proto icmp from any to 20.20.20.0/24 icmp-type 11
block in log quick on tun0 proto icmp from any to any
```

Adding these 3 rules to the anti-spoofing rules is a bit tricky. One error might be to put the new ICMP rules at the beginning:

```
pass  in     quick on tun0 proto icmp from any to 20.20.20.0/24 icmp-type 0
pass  in     quick on tun0 proto icmp from any to 20.20.20.0/24 icmp-type 11
block in log quick on tun0 proto icmp from any to any
block in     quick on tun0 from 192.168.0.0/16 to any
block in     quick on tun0 from 172.16.0.0/12 to any
block in     quick on tun0 from 10.0.0.0/8 to any
block in     quick on tun0 from 127.0.0.0/8 to any
block in     quick on tun0 from 0.0.0.0/8 to any
block in     quick on tun0 from 169.254.0.0/16 to any
block in     quick on tun0 from 192.0.2.0/24 to any
block in     quick on tun0 from 204.152.64.0/23 to any
block in     quick on tun0 from 224.0.0.0/3 to any
block in log quick on tun0 from 20.20.20.0/24 to any
block in log quick on tun0 from any to 20.20.20.0/32
block in log quick on tun0 from any to 20.20.20.255/32
pass  in       all
```

The problem with this is that an ICMP type 0 packet from 192.168.0.0/16 will get passed by the first rule, and never blocked by the fourth rule. Also, since we `quick`ly pass an ICMP ECHO_REPLY (type 0) to 20.20.20.0/24, we've just opened ourselves back up to a nasty smurf attack and nullified those last two block rules. Oops. To avoid this, we place the ICMP rules after the anti-spoofing rules:

```
block in     quick on tun0 from 192.168.0.0/16 to any
block in     quick on tun0 from 172.16.0.0/12 to any
block in     quick on tun0 from 10.0.0.0/8 to any
block in     quick on tun0 from 127.0.0.0/8 to any
block in     quick on tun0 from 0.0.0.0/8 to any
block in     quick on tun0 from 169.254.0.0/16 to any
block in     quick on tun0 from 192.0.2.0/24 to any
block in     quick on tun0 from 204.152.64.0/23 to any
block in     quick on tun0 from 224.0.0.0/3 to any
block in log quick on tun0 from 20.20.20.0/24 to any
block in log quick on tun0 from any to 20.20.20.0/32
block in log quick on tun0 from any to 20.20.20.255/32
pass  in     quick on tun0 proto icmp from any to 20.20.20.0/24 icmp-type 0
pass  in     quick on tun0 proto icmp from any to 20.20.20.0/24 icmp-type 11
block in log quick on tun0 proto icmp from any to any
pass  in       all
```

Because we block spoofed traffic before the ICMP rules are processed, a spoofed packet never makes it to the ICMP ruleset. It's very important to keep such situations in mind when merging rules.

2.12. TCP and UDP Ports; The "port" Keyword

Now that we've started blocking packets based on protocol, we can start blocking packets based on specific aspects of each protocol. The most frequently used of these aspects is the port number. Services such as rsh, rlogin, and telnet are all very convenient to have, but also hideously insecure against network sniffing and spoofing. One great compromise is to only allow the services to run internally, then block them externally. This is easy to do because rlogin, rsh, and telnet use specific TCP ports (513, 514, and 23 respectively). As such, creating rules to block them is easy:

```
block in log quick on tun0 proto tcp from any to 20.20.20.0/24 port = 513
block in log quick on tun0 proto tcp from any to 20.20.20.0/24 port = 514
block in log quick on tun0 proto tcp from any to 20.20.20.0/24 port = 23
```

Make sure all 3 are before the `pass in all` and they'll be closed off from the outside (leaving out spoofing for brevity's sake):

```
pass  in     quick on tun0 proto icmp from any to 20.20.20.0/24 icmp-type 0
pass  in     quick on tun0 proto icmp from any to 20.20.20.0/24 icmp-type 11
block in log quick on tun0 proto icmp from any to any
block in log quick on tun0 proto tcp from any to 20.20.20.0/24 port = 513
block in log quick on tun0 proto tcp from any to 20.20.20.0/24 port = 514
block in log quick on tun0 proto tcp from any to 20.20.20.0/24 port = 23
pass  in     all
```

You might also want to block 514/udp (syslog), 111/tcp & 111/udp (portmap), 515/tcp (lpd), 2049/tcp and 2049/udp (NFS), 6000/tcp (X11) and so on and so forth. You can get a complete listing of the ports being listened to by using `netstat -a` (or `lsof -i`, if you have it installed).

Blocking UDP instead of TCP only requires replacing `proto tcp` with `proto udp`. The rule for syslog would be:

```
block in log quick on tun0 proto udp from any to 20.20.20.0/24 port = 514
```

IPF also has a shorthand way to write rules that apply to both `proto tcp` and `proto udp` at the same time, such as portmap or NFS. The rule for portmap would be:

```
block in log quick on tun0 proto tcp/udp from any to 20.20.20.0/24 port = 111
```

3. Advanced Firewalling Introduction

This section is designed as an immediate followup to the basic section. Contained below are both concepts for advanced firewall design, and advanced features contained only within ipfilter. Once you are comfortable with this section, you should be able to build a very strong firewall.

3.1. Rampant Paranoia; or The Default-Deny Stance

There's a big problem with blocking services by the port: sometimes they move. RPC based programs are terrible about this, lockd, statd, even nfsd listens places other than 2049. It's awfully hard to predict, and even worse to automate adjusting all the time. What if you miss a service? Instead of dealing with all that hassle, let's start over with a clean slate. The current ruleset looks like this:

Yes, we really are starting over. The first rule we're going to use is this:

```
block in all
```

No network traffic gets through. None. Not a peep. You're rather secure with this setup. Not terribly useful, but quite secure. The great thing is that it doesn't take much more to make your box rather secure, yet useful too. Let's say the machine this is running on is a web server, nothing more, nothing less. It doesn't even do DNS lookups. It just wants to take connections on 80/tcp and that's it. We can do that. We can do that with a second rule, and you already know how:

```
block in       on tun0 all
pass  in quick on tun0 proto tcp from any to 20.20.20.1/32 port = 80
```

This machine will pass in port 80 traffic for 20.20.20.1, and deny everything else, including responses from port 80. To fix this, you have to allow the response back out as well:

```
block in       on tun0 all
pass  in quick on tun0 proto tcp from any to 20.20.20.1/32 port = 80
pass  out quick on tun0 proto tcp from 20.20.20.1/32 port = 80 to any
```

For basic firewalling, this is all one needs. However, there is an easier way; by using stateful rules.

3.2. Implicit Allow; The "keep state" Rule

The job of your firewall is to prevent unwanted traffic getting to point B from point A. We have general rules which say "as long as this packet is to port 23, it's okay." We have general rules which say "as long as this packet has its FIN flag set, it's okay." Our firewalls don't know the beginning, middle, or end of any TCP/UDP/ICMP session. They merely have vague rules that are applied to all packets. We're left to hope that the packet with its FIN flag set isn't really a FIN scan, mapping our services. We hope that the packet to port 23 isn't an attempted hijack of our telnet session. What if there was a way to identify and authorize individual TCP/UDP/ICMP sessions and distinguish them from port scanners and DoS attacks? There is a way, it's called keeping state.

We want convenience and security in one. Lots of people do, that's why Ciscos have an "established" clause that lets established tcp sessions go through. Ipfw has established. Ipfwadm has setup/established. They all have this feature, but the name is very misleading. When we first saw it, we thought it meant our packet filter was keeping track of what was going on, that it knew if a connection was really established or not. The fact is, they're all taking the packet's word for it from a part of the packet anybody can lie about. They read the TCP packet's flags section and there's the reason UDP/ICMP don't work with it, they have no such thing. Anybody who can create a packet with bogus flags can get by a firewall with this setup.

Where does IPF come in to play here, you ask? Well, unlike the other firewalls, IPF really can keep track of whether or not a connection is established. And it'll do it with TCP, UDP and ICMP, not just TCP. Ipf calls it keeping state. The keyword for the ruleset is `keep state`.

Up until now, we've told you that packets come in, then the ruleset gets checked; packets go out, then the ruleset gets checked. Actually, what happens is packets come in, the state table gets checked, then *maybe* the inbound ruleset gets checked; packets go out, the state table gets checked, then *maybe* the outbound ruleset gets checked. The state table is a list of TCP/UDP/ICMP sessions that are unquestionadely passed through the firewall, circumventing the entire ruleset. Sound like a serious security hole? Hang on, it's the best thing that ever happened to your firewall.

All TCP/IP sessions have a start, a middle, and an end (even though they're sometimes all in the same packet). You can't have an end without a middle and you can't have a middle without a start. This means that all you really need to filter on is the beginning of a TCP/UDP/ICMP session. If the beginning of the session is allowed by your firewall rules, you really want the middle and end to be allowed too (lest your IP stack should overflow and your machines become useless). Keeping state allows you to ignore the middle and end and simply focus on blocking/passing new sessions. If the new session is passed, all its subsequent packets will be allowed through. If it's blocked, none of its subsequent packets will be allowed through. Here's an example for running an ssh server (and nothing but an ssh server):

```
block out quick on tun0 all
pass  in  quick on tun0 proto tcp from any to 20.20.20.1/32 port = 22 keep state
```

The first thing you might notice is that there's no "pass out" provision. In fact, there's only an all-inclusive "block out" rule. Despite this, the ruleset is complete. This is because by keeping state, the entire ruleset is circumvented. Once the first SYN packet hits the ssh server, state is created and the remainder of the ssh session is allowed to take place without interference from the firewall. Here's another example:

```
block in  quick on tun0 all
pass  out quick on tun0 proto tcp from 20.20.20.1/32 to any keep state
```

In this case, the server is running no services. Infact, it's not a server, it's a client. And this client doesn't want unauthorized packets entering its IP stack at all. However, the client wants full access to the internet and the reply packets that such privledge entails. This simple ruleset creates state entries for every new outgoing TCP session. Again, since a state entry is created, these new TCP sessions are free to talk back and forth as they please without the hinderance or inspection of the firewall ruleset. We mentioned that this also works for UDP and ICMP:

```
block in  quick on tun0 all
pass  out quick on tun0 proto tcp  from 20.20.20.1/32 to any keep state
pass  out quick on tun0 proto udp  from 20.20.20.1/32 to any keep state
pass  out quick on tun0 proto icmp from 20.20.20.1/32 to any keep state
```

Yes Virginia, we can ping. Now we're keeping state on TCP, UDP, ICMP. Now we can make outgoing connections as though there's no firewall at all, yet would-be attackers can't get back in. This is very handy because there's no need to track down what ports we're listening to, only the ports we want people to be able to get to.

State is pretty handy, but it's also a bit tricky. You can shoot yourself in the foot in strange and mysterious ways. Consider the following ruleset:

```
pass  in  quick on tun0 proto tcp from any to 20.20.20.1/32 port = 23
pass  out quick on tun0 proto tcp from any to any keep state
block in  quick all
block out quick all
```

At first glance, this seems to be a good setup. We allow incoming sessions to port 23, and outgoing sessions anywhere. Naturally packets going to port 23 will have reply packets, but the ruleset is setup in such a way that the pass out rule will generate a state entry and everything will work perfectly. At least, you'd think so.

The unfortunate truth is that after 60 seconds of idle time the state entry will be closed (as opposed to the normal 5 days). This is because the state tracker never saw the original SYN packet destined to port 23, it only saw the SYN ACK. IPF is very good about following TCP sessions from start to finish, but it's not very good about coming into the middle of a connection, so rewrite the rule to look like this:

```
pass  in  quick on tun0 proto tcp from any to 20.20.20.1/32 port = 23 keep state
pass  out quick on tun0 proto tcp from any to any keep state
block in  quick all
block out quick all
```

The additional of this rule will enter the very first packet into the state table and everything will work as expected. Once the 3-way handshake has been witness by the state engine, it is marked in 4/4 mode, which means it's setup for long-term data exchange until such time as the connection is torn down (wherein the mode changes again. You can see the current modes of your state table with `ipfstat -s`.

3.3. Stateful UDP

UDP is stateless so naturally it's a bit harder to do a reliable job of keeping state on it. Nonetheless, ipf does a pretty good job. When machine A sends a UDP packet to machine B with source port X and destination port Y, ipf will allow a reply from machine B to machine A with source port Y and destination port X. This is a short term state entry, a mere 60 seconds.

Here's an example of what happens if we use nslookup to get the IP address of www.3com.com:

```
$ nslookup www.3com.com
```

A DNS packet is generated:

```
17:54:25.499852 20.20.20.1.2111 > 198.41.0.5.53: 51979+
```

The packet is from 20.20.20.1, port 2111, destined for 198.41.0.5, port 53. A 60 second state entry is created. If a packet comes back from 198.41.0.5 port 53 destined for 20.20.20.1 port 2111 within that period of time, the reply packet will be let through. As you can see, milliseconds later:

```
17:54:25.501209 198.41.0.5.53 > 20.20.20.1.2111: 51979 q: www.3com.com
```

The reply packet matches the state criteria and is let through. At that same moment that packet is let through, the state gateway is closed and no new incoming packets will be allowed in, even if they claim to be from the same place.

3.4. Stateful ICMP

IPFilter handles ICMP states in the manner that one would expect from understanding how ICMP is used with TCP and UDP, and with your understanding of how `keep state` works. There are two general types of ICMP messages; requests and replies. When you write a rule such as:

```
pass out on tun0 proto icmp from any to any icmp-type 8 keep state
```

to allow outbound echo requests (a typical ping), the resultant icmp-type 0 packet that comes back will be allowed in. This state entry has a default timeout of an incomplete 0/0 state of 60 seconds. Thus, if you are keeping state on any outbound icmp message that will elicit an icmp message in reply, you need a `proto icmp [...] keep state` rule.

However, the majority of ICMP messages are status messages generated by some failure in UDP (and sometimes TCP), and in 3.4.x and greater IPFilters, any ICMP error status message (say `icmp-type 3 code 3` port unreachable, or `icmp-type 11` time exceeded) that matches an active state table entry that could have generated that message, the ICMP packet is let in. For example, in older IPFilters, if you wanted traceroute to work, you needed to use:

```
pass out on tun0 proto udp from any to any port 33434><33690 keep state
pass in on tun0 proto icmp from any to any icmp-type timex
```

whereas now you can do the right thing and just keep state on udp with:

```
pass out on tun0 proto udp from any to any port 33434><33690 keep state
```

To provide some protection against a third-party sneaking ICMP messages through your firewall when an active connection is known to be in your state table, the incoming ICMP packet is checked not only for matching source and destination addresses (and ports, when applicable) but a tiny part of the payload of the packet that the ICMP message is claiming it was generated by.

3.5. FIN Scan Detection; "flags" Keyword, "keep frags" Keyword

Let's go back to the 4 rule set from the previous section:

```
pass  in  quick on tun0 proto tcp from any to 20.20.20.1/32 port = 23 keep state
pass  out quick on tun0 proto tcp from any to any keep state
block in  quick all
block out quick all
```

This is almost, but not quite, satisfactory. The problem is that it's not just SYN packets that're allowed to go to port 23, any old packet can get through. We can change this by using the `flags` option:

```
pass  in  quick on tun0 proto tcp from any to 20.20.20.1/32 port = 23 flags S keep state
pass  out quick on tun0 proto tcp from any to any flags S keep state
block in  quick all
block out quick all
```

Now only TCP packets, destined for 20.20.20.1, at port 23, with a lone SYN flag will be allowed in and entered into the state table. A lone SYN flag is only present as the very first packet in a TCP session (called the TCP handshake) and that's really what we wanted all along. There's at least two advantages to this: No arbitrary packets can come in and make a mess of your state table. Also, FIN and XMAS scans will fail since they set flags other than the SYN flag.† Now all incoming packets must either be handshakes or have state already. If anything else comes in, it's probably a port scan or a forged packet. There's one exception to that, which is when a packet comes in that's fragmented from its journey. IPF has provisions for this as well, the `keep frags` keyword. With it, IPF will notice and keep track of packets that are fragmented, allowing the expected fragments to to go through. Let's rewrite the 3 rules to log forgeries and allow fragments:

---

† Some examples use `flags S/SA` instead of `flags S`. `flags S` actually equates to `flags S/AUPRFS` and matches against only the SYN packet out of all six possible flags, while `flags S/SA` will allow packets that may or may not have the URG, PSH, FIN, or RST flags set. Some protocols demand the URG or PSH flags, and `S/SAFR` would be a better choice for these, however we feel that it is less secure to blindly use `S/SA` when it isn't required. But it's your firewall.

```
pass  in  quick on tun0 proto tcp from any to 20.20.20.1/32 port = 23 flags S keep state keep frags
pass  out quick on tun0 proto tcp from any to any keep state flags S keep frags
block in  log quick all
block out log quick all
```

This works because every packet that should be allowed through makes it into the state table before the blocking rules are reached. The only scan this won't detect is a SYN scan itself. If you're truely worried about that, you might even want to log all initial SYN packets.

3.6. Responding To a Blocked Packet

So far, all of our blocked packets have been dumped on the floor, logged or not, we've never sent anything back to the originating host. Sometimes this isn't the most desirable of responses because in doing so, we actually tell the attacker that a packet filter is present. It seems a far better thing to misguide the attacker into believing that, while there's no packet filter running, there's likewise no services to break into. This is where fancier blocking comes into play.

When a service isn't running on a Unix system, it normally lets the remote host know with some sort of return packet. In TCP, this is done with an RST (Reset) packet. When blocking a TCP packet, IPF can actually return an RST to the origin by using the `return-rst` keyword.

Where once we did:

```
block in log on tun0 proto tcp from any to 20.20.20.0/24 port = 23
pass  in     all
```

We might now do:

```
block return-rst in log proto tcp from any to 20.20.20.0/24 port = 23
block in log quick on tun0
pass  in     all
```

We need two `block` statements since `return-rst` only works with TCP, and we still want to block protocols such as UDP, ICMP, and others. Now that this is done, the remote side will get "connection refused" instead of "connection timed out".

It's also possible to send an error message when somebody sends a packet to a UDP port on your system. Whereas once you might have used:

```
block in log quick on tun0 proto udp from any to 20.20.20.0/24 port = 111
```

You could instead use the `return-icmp` keyword to send a reply:

```
block return-icmp(port-unr) in log quick on tun0 proto udp from any to 20.20.20.0/24 port = 111
```

According to *TCP/IP Illustrated*, port-unreachable is the correct ICMP type to return when no service is listening on the port in question. You can use any ICMP type you like, but port-unreachable is probably your best bet. It's also the default ICMP type for `return-icmp`.

However, when using `return-icmp`, you'll notice that it's not very stealthy, and it returns the ICMP packet with the IP address of the firewall, not the original destination of the packet. This was fixed in ipfilter 3.3, and a new keyword; `return-icmp-as-dest`, has been added. The new format is:

```
block return-icmp-as-dest(port-unr) in log on tun0 proto udp from any to 20.20.20.0/24 port = 111
```

Additionally, you should be careful to use response packets only in situations where you understand ahead of time what it is that you're responding to. For example, if you happened to return-icmp to a local network broadcast address, you would end up flooding your network in short order.†

3.7. Fancy Logging Techniques

It is important to note that the presence of the `log` keyword only ensures that the packet will be available to the ipfilter logging device; `/dev/ipl`. In order to actually see this log information, one must be running the `ipmon` utility (or some other utility that reads from `/dev/ipl`). The typical usage of `log`

† This is especially true in a DHCP/BOOTP environment such as a typical consumer broadband connection where you run the risk of "attacking" your own ISP's DHCP server. They won't appreciate it.

is coupled with `ipmon -s` to log the information to syslog. As of ipfilter 3.3, one can now even control the logging behavior of syslog by using `log level` keywords, as in rules such as this:

```
block in log level auth.info quick on tun0 from 20.20.20.0/24 to any
block in log level auth.alert quick on tun0 proto tcp from any to 20.20.20.0/24 port = 21
```

In addition to this, you can tailor what information is being logged. For example, you may not be interested that someone attempted to probe your telnet port 500 times, but you are interested that they probed you once. You can use the `log first` keyword to only log the first example of a packet. Of course, the notion of "first-ness" only applies to packets in a specific session, and for the typical blocked packet, you will be hard pressed to encounter situations where this does what you expect. However, if used in conjunction with `pass` and `keep state`, this can be a valuable keyword for keeping tabs on traffic.

Another useful thing you can do with the logs is to keep track of interesting pieces of the packet in addition to the header information normally being logged. Ipfilter will give you the first 128 bytes of the packet if you use the `log body` keyword. You should limit the use of body logging, as it makes your logs very verbose, but for certain applications, it is often handy to be able to go back and take a look at the packet, or to send this data to another application that can examine it further.

3.8. Putting It All Together

So now we have a pretty tight firewall, but it can still be tighter. Some of the original ruleset we wiped clean is actually very useful. I'd suggest bringing back all the anti-spoofing stuff. This leaves us with:

```
block in           on tun0
block in     quick on tun0 from 192.168.0.0/16 to any
block in     quick on tun0 from 172.16.0.0/12 to any
block in     quick on tun0 from 10.0.0.0/8 to any
block in     quick on tun0 from 127.0.0.0/8 to any
block in     quick on tun0 from 0.0.0.0/8 to any
block in     quick on tun0 from 169.254.0.0/16 to any
block in     quick on tun0 from 192.0.2.0/24 to any
block in     quick on tun0 from 204.152.64.0/23 to any
block in     quick on tun0 from 224.0.0.0/3 to any
block in log quick on tun0 from 20.20.20.0/24 to any
block in log quick on tun0 from any to 20.20.20.0/32
block in log quick on tun0 from any to 20.20.20.255/32
pass  out quick on tun0 proto tcp/udp from 20.20.20.1/32 to any keep state
pass  out quick on tun0 proto icmp    from 20.20.20.1/32 to any keep state
pass  in  quick on tun0 proto tcp from any to 20.20.20.1/32 port = 80 flags S keep state
```

3.9. Improving Performance With Rule Groups

Let's extend our use of our firewall by creating a much more complicated, and we hope more applicable to the real world, example configuration For this example, we're going to change the interface names, and network numbers. Let's assume that we have three interfaces in our firewall with interfaces `xl0`, `xl1`, and `xl2`.

```
xl0 is connected to our external network 20.20.20.0/26
xl1 is connected to our "DMZ" network 20.20.20.64/26
xl2 is connected to our protected network 20.20.20.128/25
```

We'll define the entire ruleset in one swoop, since we figure that you can read these rules by now:

```
block in     quick on xl0 from 192.168.0.0/16 to any
block in     quick on xl0 from 172.16.0.0/12 to any
block in     quick on xl0 from 10.0.0.0/8 to any
block in     quick on xl0 from 127.0.0.0/8 to any
block in     quick on xl0 from 0.0.0.0/8 to any
block in     quick on xl0 from 169.254.0.0/16 to any
block in     quick on xl0 from 192.0.2.0/24 to any
block in     quick on xl0 from 204.152.64.0/23 to any
block in     quick on xl0 from 224.0.0.0/3 to any
block in log quick on xl0 from 20.20.20.0/24 to any
block in log quick on xl0 from any to 20.20.20.0/32
block in log quick on xl0 from any to 20.20.20.63/32
block in log quick on xl0 from any to 20.20.20.64/32
block in log quick on xl0 from any to 20.20.20.127/32
block in log quick on xl0 from any to 20.20.20.128/32
```

```
block in log quick on xl0 from any to 20.20.20.255/32
pass out on xl0 all

pass out quick on xl1 proto tcp from any to 20.20.20.64/26 port = 80 flags S keep state
pass out quick on xl1 proto tcp from any to 20.20.20.64/26 port = 21 flags S keep state
pass out quick on xl1 proto tcp from any to 20.20.20.64/26 port = 20 flags S keep state
pass out quick on xl1 proto tcp from any to 20.20.20.65/32 port = 53 flags S keep state
pass out quick on xl1 proto udp from any to 20.20.20.65/32 port = 53 keep state
pass out quick on xl1 proto tcp from any to 20.20.20.66/32 port = 53 flags S keep state
pass out quick on xl1 proto udp from any to 20.20.20.66/32 port = 53 keep state
block out on xl1 all
pass in quick on xl1 proto tcp/udp from 20.20.20.64/26 to any keep state

block out on xl2 all
pass in quick on xl2 proto tcp/udp from 20.20.20.128/25 to any keep state
```

From this arbitarary example, we can already see that our ruleset is becoming unwieldy. To make matters worse, as we add more specific rules to our DMZ network, we add additional tests that must be parsed for every packet, which affects the performance of the `xl0` <-> `xl2` connections. If you set up a firewall with a ruleset like this, and you have lots of bandwidth and a moderate amount of cpu, everyone that has a workstation on the `xl2` network is going to come looking for your head to place on a platter. So, to keep your head <-> torso network intact, you can speed things along by creating rule groups. Rule groups allow you to write your ruleset in a tree fashion, instead of as a linear list, so that if your packet has nothing to do with the set of tests (say, all those `xl1` rules) those rules will never be consulted. It's somewhat like having multiple firewalls all on the same machine.

Here's a simple example to get us started:

```
block out quick on xl1 all head 10
pass out quick proto tcp from any to 20.20.20.64/26 port = 80 flags S keep state group 10
block out on xl2 all
```

In this simplistic example, we can see a small hint of the power of the rule group. If the packet is not destined for `xl1`, the `head` of rule `group 10` will not match, and we will go on with our tests. If the packet does match for `xl1`, the `quick` keyword will short-circuit all further processing at the root level (rule group 0), and focus the testing on rules which belong to `group 10`; namely, the SYN check for 80/tcp. In this way, we can re-write the above rules so that we can maximize performance of our firewall.

```
block in quick on xl0 all head 1
  block in     quick on xl0 from 192.168.0.0/16 to any  group 1
  block in     quick on xl0 from 172.16.0.0/12 to any   group 1
  block in     quick on xl0 from 10.0.0.0/8 to any      group 1
  block in     quick on xl0 from 127.0.0.0/8 to any     group 1
  block in     quick on xl0 from 0.0.0.0/8 to any       group 1
  block in     quick on xl0 from 169.254.0.0/16 to any  group 1
  block in     quick on xl0 from 192.0.2.0/24 to any    group 1
  block in     quick on xl0 from 204.152.64.0/23 to any group 1
  block in     quick on xl0 from 224.0.0.0/3 to any     group 1
  block in log quick on xl0 from 20.20.20.0/24 to any   group 1
  block in log quick on xl0 from any to 20.20.20.0/32   group 1
  block in log quick on xl0 from any to 20.20.20.63/32  group 1
  block in log quick on xl0 from any to 20.20.20.64/32  group 1
  block in log quick on xl0 from any to 20.20.20.127/32 group 1
  block in log quick on xl0 from any to 20.20.20.128/32 group 1
  block in log quick on xl0 from any to 20.20.20.255/32 group 1
  pass  in           on xl0 all group 1

pass out on xl0 all

block out quick on xl1 all head 10
  pass out quick on xl1 proto tcp from any to 20.20.20.64/26 port = 80 flags S keep state group 10
  pass out quick on xl1 proto tcp from any to 20.20.20.64/26 port = 21 flags S keep state group 10
  pass out quick on xl1 proto tcp from any to 20.20.20.64/26 port = 20 flags S keep state group 10
  pass out quick on xl1 proto tcp from any to 20.20.20.65/32 port = 53 flags S keep state group 10
  pass out quick on xl1 proto udp from any to 20.20.20.65/32 port = 53         keep state group 10
  pass out quick on xl1 proto tcp from any to 20.20.20.66/32 port = 53 flags S keep state
  pass out quick on xl1 proto udp from any to 20.20.20.66/32 port = 53         keep state group 10

pass in quick on xl1 proto tcp/udp from 20.20.20.64/26 to any keep state

block out on xl2 all

pass in quick on xl2 proto tcp/udp from 20.20.20.128/25 to any keep state
```

Now you can see the rule groups in action. For a host on the `xl2` network, we can completely bypass all the checks in `group 10` when we're not communicating with hosts on that network.

Depending on your situation, it may be prudent to group your rules by protocol, or various machines, or netblocks, or whatever makes it flow smoothly.

3.10. "Fastroute"; The Keyword of Stealthiness

Even though we're forwarding some packets, and blocking other packets, we're typically behaving like a well behaved router should by decrementing the TTL on the packet and acknowledging to the entire world that yes, there is a hop here. But we can hide our presence from inquisitive applications like unix traceroute which uses UDP packets with various TTL values to map the hops between two sites. If we want incoming traceroutes to work, but we do not want to announce the presence of our firewall as a hop, we can do so with a rule like this:

```
block in quick on xl0 fastroute proto udp from any to any port 33434 >< 33465
```

The presence of the `fastroute` keyword will signal ipfilter to not pass the packet into the Unix IP stack for routing which results in a TTL decrement. The packet will be placed gently on the output interface by ipfilter itself and no such decrement will happen. Ipfilter will of course use the system's routing table to figure out what the appropriate output interface really is, but it will take care of the actual task of routing itself.

There's a reason we used `block quick` in our example, too. If we had used `pass`, and if we had IP Forwarding enabled in our kernel, we would end up having two paths for a packet to come out of, and we would probably panic our kernel.

It should be noted, however, that most Unix kernels (and certainly the ones underlying the systems that ipfilter usually runs on) have far more efficient routing code than what exists in ipfilter, and this keyword should not be thought of as a way to improve the operating speed of your firewall, and should only be used in places where stealth is an issue.

4. NAT and Proxies

Outside of the corporate environment, one of the biggest enticements of firewall technology to the end user is the ability to connect several computers through a common external interface, often without the approval, knowledge, or even consent of their service provider. To those familiar with Linux, this concept is called *IP Masquerading*, but to the rest of the world it is known by the more obscure name of *Network Address Translation,* or NAT for short.†

4.1. Mapping Many Addresses Into One Address

The basic use of NAT accomplishes much the same thing that Linux's IP Masquerading function does, and it does it with one simple rule:

```
map tun0 192.168.1.0/24 -> 20.20.20.1/32
```

Very simple. Whenever a packet goes out the `tun0` interface with a source address matching the CIDR network mask of 192.168.1.0/24 †† this packet will be rewritten within the IP stack such that its source address is 20.20.20.1, and it will be sent on to its original destination. The system also keeps a list of what translated connections are in progress so that it can perform the reverse and remap the response (which will be directed to 20.20.20.1) to the internal host that really generated the packet.

---

† To be pedantic, what IPFilter provides is really called NPAT, for Network and Port Address Translation, which means we can change any of the source and destination IP Addresses and their source and destination ports. *True NAT* only allows one to change the addresses.

†† This is a typical internal address space, since it's non-routable on the Real Internet it is often used for internal networks. You should still block these packets coming in from the outside world as discussed earlier.

There is a drawback to the rule we have just written, though. In a large number of cases, we do not happen to know what the IP address of our outside link is (if we're using tun0 or ppp0 and a typical ISP) so it makes setting up our NAT tables a chore. Luckily, NAT is smart enough to accept an address of 0/32 as a signal that it needs to go look at what the address of that interface really is and we can rewrite our rule as follows:

```
map tun0 192.168.1.0/24 -> 0/32
```

Now we can load our ipnat rules with impunity and connect to the outside world without having to edit anything. You do have to run ipf -y to refresh the address if you get disconnected and redial or if your DHCP lease changes, though.†††

Some of you may be wondering what happens to the source port when the mapping happens. With our current rule, the packet's source port is unchanged from the original source port. There can be instances where we do not desire this behavior; maybe we have another firewall further upstream we have to pass through, or perhaps many hosts are trying to use the same source port, causing a collision where the rule doesn't match and the packet is passed untranslated. ipnat helps us here with the portmap keyword:

```
map tun0 192.168.1.0/24 -> 0/32 portmap tcp/udp 20000:30000
```

Our rule now shoehorns all the translated connections (which can be tcp, udp, or tcp/udp) into the port range of 20000 to 30000.

Additionally, we can make our lives even easier by using the auto keyword to tell ipnat to determine for itself which ports are available for use, and allocate a proportional amount of them per address in your pool versus addresses being natted:

```
map tun0 192.168.1.0/24 -> 0/32 portmap tcp/udp auto
```

Keep in mind that these portmap rules only apply to the protocols that you have specified (e.g.: tcp, udp, or tcp/udp), and do not apply to other protocols like ICMP or IPSec ESP/AH. For these, you need to have an additonal map statement that applies to all other protocols:

```
map tun0 192.168.1.0/24 -> 0/32 portmap tcp/udp 20000:30000
map tun0 192.168.1.0/24 -> 0/32
```

4.2. Mapping Many Addresses Into a Pool of Addresses

Another use common use of NAT is to take a small statically allocated block of addresses and map many computers into this smaller address space. This is easy to accomplish using what you already know about the map and portmap keywords by writing a rule like so:

```
map tun0 192.168.0.0/16 -> 20.20.20.0/24 portmap tcp/udp 20000:60000
```

Also, there may be instances where a remote application requires that multiple connections all come from the same IP address. We can help with these situations by telling NAT to statically map sessions from a host into the pool of addresses and work some magic to choose a port. This uses a the keyword map-block as follows:

```
map-block tun0 192.168.1.0/24 -> 20.20.20.0/24
```

As with map, map-block has it's own version of portmap. This is accomplished with either the auto keyword:

```
map-block tun0 192.168.1.0/24 -> 20.20.20.0/24 auto
```

or the ports keyword, if you want to specify your own number of ports belonging to each ip address:

```
map-block tun0 192.168.1.0/24 -> 20.20.20.0/24 ports 64
```

††† Typically you'd do this in your network interface setup scripts for your ppp client, but that is well beyond the scope of this document.

4.3. One to One Mappings

Occasionally it is desirable to have a system with one IP address behind the firewall to appear to have a completely different IP address. One example of how this would work would be a lab of computers which are then attached to various networks that are to be put under some kind of test. In this example, you would not want to have to reconfigure the entire lab when you could place a NAT system in front and change the addresses in one simple place. We can do that with the `bimap` keyword, for bidirectional mapping. `Bimap` has some additional protections on it to ensure a known state for the connection, whereas the `map` keyword is designed to allocate an address and a source port and rewrite the packet and go on with life.

```
bimap tun0 192.168.1.1/32 -> 20.20.20.1/32
```

will accomplish the mapping for one host.

4.4. Policy NAT

Quite often, we will find ourselves in need of having NAT behave differently based on which source and destination addresses are being used. For example, we might want to always NAT unless we are speaking with a specific subnet:

```
map tun0 from 192.168.1.0/24 ! to 5.0.0.0/20 -> 20.20.20.1/32
```

We might also want to do something like the following:

```
map tun0 from 192.168.1.5/32 port = 5555 to 1.2.3.4/32 -> 20.20.20.2/32
map tun0 from 192.168.1.0/24 to 5.0.0.0/20 -> 20.20.20.2/32 portmap auto
```

4.5. Spoofing Services

Spoofing services? What does that have to do with anything? Plenty. Let's pretend that we have a web server running on 20.20.20.5, and since we've gotten increasingly suspicious of our network security, we desire to not run this server on port 80 since that requires a brief lifespan as the root user. But how do we run it on a less privledged port of 8000 in this world of "anything dot com"? How will anyone find our server? We can use the redirection facilities of NAT to solve this problem by instructing it to remap any connections destined for 20.20.20.5:80 to really point to 20.20.20.5:8000. This uses the `rdr` keyword:

```
rdr tun0 20.20.20.5/32 port 80 -> 192.168.0.5 port 8000
```

We can also specify the protocol here, if we wanted to redirect a UDP service, instead of a TCP service (which is the default). For example, if we had a honeypot on our firewall to impersonate the popular Back Orifice for Windows, we could shovel our entire network into this one place with a simple rule:

```
rdr tun0 20.20.20.0/24 port 31337 -> 127.0.0.1 port 31337 udp
```

We can also alter `rdr`'s behavior based on the source and destination addresses:

```
rdr tun0 from 10.1.1.1/32 to 20.20.20.5/32 port = 80 -> 192.168.0.5 port 8001
rdr tun0 from 10.1.1.1/32 port = 12345 to 20.20.20.5/32 port = 80 -> 192.168.0.5 port 8002
```

An extremely important point must be made about `rdr`: You cannot easily† use this feature as a "reflector". E.g:

```
rdr tun0 20.20.20.5/32 port 80 -> 20.20.20.6 port 80 tcp
```

will not work in the situation where .5 and .6 are on the same LAN segment. The `rdr` function is applied to packets that enter the firewall on the specified interface. When a packet comes in that matches a `rdr` rule, its destination address is then rewritten, it is pushed into `ipf` for filtering, and should it successfully run the gauntlet of filter rules, it is then sent to the unix routing code. Since this packet is still *inbound* on the same interface that it will need to leave the system on to reach a host, the system gets confused. Reflectors don't work. Neither does specifying the address of the interface the packet just came in on. Always

† Yes. There is a way to do this. It's so convoluted that I refuse to use it, though. Smart people who require this functionality will transparently redirect into something like TIS plug-gw on 127.0.0.1. Stupid people will set up a dummy loop interface pair and double rewrite.

remember that `rdr` destinations must exit out of the firewall host on a different interface. ††

4.6. Transparent Proxy Support; Redirection Made Useful

Since you're installing a firewall, you may have decided that it is prudent to use a proxy for many of your outgoing connections so that you can further tighten your filter rules protecting your internal network, or you may have run into a situation that the NAT mapping process does not currently handle properly. This can also be accomplished with a redirection statement:

```
rdr xl0 0.0.0.0/0 port 21 -> 127.0.0.1 port 21
```

This statement says that any packet coming in on the `xl0` interface destined for any address (0.0.0.0/0) on the ftp port should be rewritten to connect it with a proxy that is running on the NAT system on port 21.

This specific example of FTP proxying does lead to some complications when used with web browsers or other automatic-login type clients that are unaware of the requirements of communicating with the proxy. There are patches for *TIS Firewall Toolkit's*`ftp-gw` to mate it with the nat process so that it can determine where you were trying to go and automatically send you there. Many proxy packages now work in a transparent proxy environment (Squid for example, located at *http://squid.nlanr.net*, works fine.)

This application of the `rdr` keyword is often more useful when you wish to force users to authenticate themselves with the proxy. (For example, you desire your engineers to be able to surf the web, but you would rather not have your call-center staff doing so.)

4.7. Filtering Redirected Services

Quite a lot of users will want to combine both filtering and address translation in order to provide a service to only known hosts behind their NAT system. For example, to provide a web server behind your machine 20.20.20.5 (which is really 192.168.0.5 on your inside network) for your friend over on 172.16.8.2, you would write the following in ipnat.rules:

```
rdr tun0 20.20.20.5/32 port 80 -> 192.168.0.5 port 8000
```

and the following in ipf.rules:

```
pass in on tun0 proto tcp from 172.16.8.2/32 to 192.168.0.5/32 port = 8000 flags S keep state
```

That might look a little strange at first glance, but that's because the NAT stage happens first, and the packet's destination address and port is rewritten before it is processed by the filter code.

4.8. Magic Hidden Within NAT; Application Proxies

Since `ipnat` provides a method to rewrite packets as they traverse the firewall, it becomes a convenient place to build in some application level proxies to make up for well known deficiencies of that application and typical firewalls. For example; FTP. We can make our firewall pay attention to the packets going across it and when it notices that it's dealing with an Active FTP session, it can write itself some temporary rules, much like what happens with `keep state`, so that the FTP data connection works. To do this, we use a rule like so:

```
map tun0 192.168.1.0/24 -> 20.20.20.1/32 proxy port ftp ftp/tcp
```

You must always remember to place this `proxy rule` **before** any `portmap` rules, otherwise when `portmap` comes along and matches the packet and rewrites it before the proxy gets a chance to work on it. Remember that ipnat rules are first-match.

There also exist proxies for "rcmd" (which we suspect is berkeley r-* commands which should be forbidden anyway, thus we haven't looked at what this proxy does) and "raudio" for Real Audio PNM streams. Likewise, both of these rules should be put before any `portmap` rules, if you're doing NAT.

---

†† This includes 127.0.0.1, by the way. That's on lo0. Neat, huh?

4.9. More Magic; Using NAT As a Load Balancer

Since `ipnat` is already rewriting packets for us, we can use it to support one of the simpler features of those expensive load-balancing systems and assign sessions to multiple destination addresses. This is accomplished using the `round-robin` keyword:

```
rdr tun0 20.20.20.5/32 port 80 -> 192.168.0.5, 192.168.0.6, 192.168.0.7 port 8000
```

5. Loading and Manipulating Filter Rules; The ipf Utility

IP Filter rules are loaded by using the `ipf` utility. The filter rules can be stored in any file on the system, but typically these rules are stored in `/etc/ipf.rules`, `/usr/local/etc/ipf.rules`, or `/etc/opt/ipf/ipf.rules`.

IP Filter has two sets of rules, the *active set* and the *inactive set*. By default, all operations are performed on the active set. You can manipulate the inactive set by adding `-I` to the `ipf` command line. The two sets can be toggled by using the `-s` command line option. This is very useful for testing new rule sets without wiping out the old rule set.

Rules can also be removed from the list instead of added by using the `-r` command line option, but it is generally a safer idea to flush the rule set that you're working on with `-F` and completely reload it when making changes.

In summary, the easiest way to load a rule set is `ipf -Fa -f /etc/ipf.rules`. For more complicated manipulations of the rule set, please see the `ipf(1)` man page.

6. Loading and Manipulating NAT Rules; The ipnat Utility

NAT rules are loaded by using the `ipnat` utility. The NAT rules can be stored in any file on the system, but typically these rules are stored in `/etc/ipnat.rules`, `/usr/local/etc/ipnat.rules`, or `/etc/opt/ipf/ipnat.rules`.

Rules can also be removed from the list instead of added by using the `-r` command line option, but it is generally a safer idea to flush the rule set that you're working on with `-C` and completely reload it when making changes. Any active mappings are not affected by `-C`, and can be removed with `-F`.

NAT rules and active mappings can be examined with the `-l` command line option.

In summary, the easiest way to load a NAT rule set is `ipnat -CF -f /etc/ipnat.rules`.

7. Monitoring and Debugging

There will come a time when you are interested in what your firewall is actually doing, and ipfilter would be incomplete if it didn't have a full suite of status monitoring tools.

7.1. The ipfstat utility

In its simplest form, `ipfstat` displays a table of interesting data about how your firewall is performing, such as how many packets have been passed or blocked, if they were logged or not, how many state entries have been made, and so on. Here's an example of something you might see from running the tool:

```
# ipfstat
 input packets:         blocked 99286 passed 1255609 nomatch 14686 counted 0
output packets:         blocked 4200 passed 1284345 nomatch 14687 counted 0
 input packets logged:  blocked 99286 passed 0
output packets logged:  blocked 0 passed 0
 packets logged:        input 0 output 0
 log failures:          input 3898 output 0
fragment state(in):     kept 0  lost 0
fragment state(out):    kept 0  lost 0
packet state(in):       kept 169364     lost 0
packet state(out):      kept 431395     lost 0
ICMP replies:   0       TCP RSTs sent:  0
```

```
Result cache hits(in):  1215208 (out):  1098963
IN Pullups succeeded:   2       failed: 0
OUT Pullups succeeded:  0       failed: 0
Fastroute successes:    0       failures:       0
TCP cksum fails(in):    0       (out):  0
Packet log flags set: (0)
        none
```

`ipfstat` is also capable of showing you your current rule list. Using the `-i` or the `-o` flag will show the currently loaded rules for in or out, respectively. Adding a `-h` to this will provide more useful information at the same time by showing you a "hit count" on each rule. For example:

```
# ipfstat -ho
2451423 pass out on xl0 from any to any
354727 block out on ppp0 from any to any
430918 pass out quick on ppp0 proto tcp/udp from 20.20.20.0/24 to any keep state keep frags
```

From this, we can see that perhaps there's something abnormal going on, since we've got a lot of blocked packets outbound, even with a very permissive `pass out` rule. Something here may warrant further investigation, or it may be functioning perfectly by design. ipfstat can't tell you if your rules are right or wrong, it can only tell you what is happening because of your rules.

To further debug your rules, you may want to use the `-n` flag, which will show the rule number next to each rule.

```
# ipfstat -on
@1 pass out on xl0 from any to any
@2 block out on ppp0 from any to any
@3 pass out quick on ppp0 proto tcp/udp from 20.20.20.0/24 to any keep state keep frags
```

The final piece of really interesting information that `ipfstat` can provide us is a dump of the state table. This is done with the `-s` flag:

```
# ipfstat -s
        281458 TCP
        319349 UDP
        0 ICMP
        19780145 hits
        5723648 misses
        0 maximum
        0 no memory
        1 active
        319349 expired
        281419 closed
100.100.100.1 -> 20.20.20.1 ttl 864000 pass 20490 pr 6 state 4/4
        pkts 196 bytes 17394    987 -> 22 585538471:2213225493 16592:16500
        pass in log quick keep state
        pkt_flags & b = 2,              pkt_options & ffffffff = 0
        pkt_security & ffff = 0, pkt_auth & ffff = 0
```

Here we see that we have one state entry for a TCP connection. The output will vary slightly from version to version, but the basic information is the same. We can see in this connection that we have a fully established connection (represented by the 4/4 state. Other states are incomplete and will be documented fully later.) We can see that the state entry has a time to live of 240 hours, which is an absurdly long time, but is the default for an established TCP connection. This TTL counter is decremented every second that the state entry is not used, and will finally result in the connection being purged if it has been left idle. The TTL is also reset to 864000 whenever the state IS used, ensuring that the entry will not time out while it is being actively used. We can also see that we have passed 196 packets consisting of about 17kB worth of data over this connection. We can see the ports for both endpoints, in this case 987 and 22; which means that this state entry represents a connection from 100.100.100.1 port 987 to 20.20.20.1 port 22. The really big numbers in the second line are the TCP sequence numbers for this connection, which helps to ensure that someone isn't easily able to inject a forged packet into your session. The TCP window is also shown. The third line is a synopsis of the implicit rule that was generated by the `keep state` code, showing that this connection is an inbound connection.

7.2. The ipmon utility

`ipfstat` is great for collecting snapshots of what's going on on the system, but it's often handy to have some kind of log to look at and watch events as they happen in time. `ipmon` is this tool. `ipmon` is capable of watching the packet log (as created with the `log` keyword in your rules), the state log, or the nat log, or any combination of the three. This tool can either be run in the foreground, or as a daemon which logs to syslog or a file. If we wanted to watch the state table in action, `ipmon -o S` would show this:

```
# ipmon -o S
01/08/1999 15:58:57.836053 STATE:NEW 100.100.100.1,53 -> 20.20.20.15,53 PR udp
01/08/1999 15:58:58.030815 STATE:NEW 20.20.20.15,123 -> 128.167.1.69,123 PR udp
01/08/1999 15:59:18.032174 STATE:NEW 20.20.20.15,123 -> 128.173.14.71,123 PR udp
01/08/1999 15:59:24.570107 STATE:EXPIRE 100.100.100.1,53 -> 20.20.20.15,53 PR udp Pkts 4 Bytes 356
01/08/1999 16:03:51.754867 STATE:NEW 20.20.20.13,1019 -> 100.100.100.10,22 PR tcp
01/08/1999 16:04:03.070127 STATE:EXPIRE 20.20.20.13,1019 -> 100.100.100.10,22 PR tcp Pkts 63 Bytes 4604
```

Here we see a state entry for an external dns request off our nameserver, two xntp pings to well-known time servers, and a very short lived outbound ssh connection.

`ipmon` is also capable of showing us what packets have been logged. For example, when using state, you'll often run into packets like this:

```
# ipmon -o I
15:57:33.803147 ppp0 @0:2 b 100.100.100.103,443 -> 20.20.20.10,4923 PR tcp len 20 1488 -A
```

What does this mean? The first field is obvious, it's a timestamp. The second field is also pretty obvious, it's the interface that this event happened on. The third field `@0:2` is something most people miss. This is the rule that caused the event to happen. Remember `ipfstat -in`? If you wanted to know where this came from, you could look there for rule 2 in rule group 0. The fourth field, the little "b" says that this packet was blocked, and you'll generally ignore this unless you're logging passed packets as well, which would be a little "p" instead. The fifth and sixth fields are pretty self-explanatory, they say where this packet came from and where it was going. The seventh ("PR") and eighth fields tell you the protocol and the ninth field tells you the size of the packet. The last part, the "-A" in this case, tells you the flags that were on the packet; This one was an ACK packet. Why did I mention state earlier? Due to the often laggy nature of the Internet, sometimes packets will be regenerated. Sometimes, you'll get two copies of the same packet, and your state rule which keeps track of sequence numbers will have already seen this packet, so it will assume that the packet is part of a different connection. Eventually this packet will run into a real rule and have to be dealt with. You'll often see the last packet of a session being closed get logged because the `keep state` code has already torn down the connection before the last packet has had a chance to make it to your firewall. This is normal, do not be alarmed. † Another example packet that might be logged:

```
12:46:12.470951 xl0 @0:1 S 20.20.20.254 -> 255.255.255.255 PR icmp len 20 9216 icmp 9/0
```

This is a ICMP router discovery broadcast. We can tell by the ICMP type 9/0.

Finally, `ipmon` also lets us look at the NAT table in action.

```
# ipmon -o N
01/08/1999 05:30:02.466114 @2 NAT:RDR 20.20.20.253,113 <- -> 20.20.20.253,113 [100.100.100.13,45816]
01/08/1999 05:30:31.990037 @2 NAT:EXPIRE 20.20.20.253,113 <- -> 20.20.20.253,113 [100.100.100.13,45816] Pkts 10 Bytes 455
```

This would be a redirection to an identd that lies to provide ident service for the hosts behind our NAT, since they are typically unable to provide this service for themselves with ordinary natting.

8. Specific Applications of IP Filter - Things that don't fit, but should be mentioned anyway.

8.1. Keep State With Servers and Flags.

Keeping state is a good thing, but it's quite easy to make a mistake in the direction that you want to `keep state` in. Generally, you want to have a `keep state` keyword on the first rule that interacts with a packet for the connection. One common mistake that is made when mixing state tracking with filtering on flags is this:

† For a technical presentation of the IP Filter stateful inspection engine, please see the white paper *Real Stateful TCP Packet Filtering in IP Filter*, by Guido van Rooij. This paper may be found at <*http://www.iae.nl/users/guido/papers/tcp_filtering.ps.gz*>

```
block in all
pass in quick proto tcp from any to 20.20.20.20/32 port = 23 flags S
pass out all keep state
```

That certainly appears to allow a connection to be created to the telnet server on 20.20.20.20, and the replies to go back. If you try using this rule, you'll see that it does work--Momentarily. Since we're filtering for the SYN flag, the state entry never fully gets completed, and the default time to live for an incomplete state is 60 seconds.

We can solve this by rewriting the rules in one of two ways:

1)

```
block in all
pass in quick proto tcp from any to 20.20.20.20/32 port = 23 keep state
block out all
```

or:

2)

```
block in all
pass in quick proto tcp from any to 20.20.20.20/32 port = 23 flags S keep state
pass out all keep state
```

Either of these sets of rules will result in a fully established state entry for a connection to your server.

8.2. Coping With FTP

FTP is one of those protocols that you just have to sit back and ask "What the heck were they thinking?" FTP has many problems that the firewall administrator needs to deal with. What's worse, the problems the administrator must face are different between making ftp clients work and making ftp servers work.

Within the FTP protocol, there are two forms of data transfer, called active and passive. Active transfers are those where the server connects to an open port on the client to send data. Conversely, passive transfers are those where the client connects to the server to receive data.

8.2.1. Running an FTP Server

In running an FTP server, handling Active FTP sessions is easy to setup. At the same time, handling Passive FTP sessions is a big problem. First we'll cover how to handle Active FTP, then move on to Passive. Generally, we can handle Active FTP sessions like we would an incoming HTTP or SMTP connection; just open the ftp port and let `keep state` do the rest:

```
pass in quick proto tcp from any to 20.20.20.20/32 port = 21 flags S keep state
pass out proto tcp all keep state
```

These rules will allow Active FTP sessions, the most common type, to your ftp server on 20.20.20.20.

The next challenge becomes handling Passive FTP connections. Web browsers default to this mode, so it's becoming quite popular and as such it should be supported. The problem with passive connections are that for every passive connection, the server starts listening on a new port (usually above 1023). This is essentially like creating a new unknown service on the server. Assuming we have a good firewall with a default-deny policy, that new service will be blocked, and thus Passive FTP sessions are broken. Don't despair! There's hope yet to be had.

A person's first inclination to solving this problem might be to just open up all ports above 1023. In truth, this will work:

```
pass in quick proto tcp from any to 20.20.20.20/32 port > 1023 flags S keep state
pass out proto tcp all keep state
```

This is somewhat unsatisfactory, though. By letting everything above 1023 in, we actually open ourselves up for a number of potential problems. While 1-1023 is the designated area for server services to run, numerous programs decided to use numbers higher than 1023, such as nfsd and X.

The good news is that your FTP server gets to decide which ports get assigned to passive sessions. This means that instead of opening all ports above 1023, you can allocate ports 15001-19999 as ftp ports and only open that range of your firewall up. In wu-ftpd, this is done with the `passive ports` option in `ftpaccess`. Please see the man page on `ftpaccess` for details in wu-ftpd configuration. On the ipfilter side, all we need do is setup corresponding rules:

```
pass in quick proto tcp from any to 20.20.20.20/32 port 15000 >< 20000 flags S keep state
pass out proto tcp all keep state
```

If even this solution doesn't satisfy you, you can always hack IPF support into your FTP server, or FTP server support into IPF.

8.2.2. Running an FTP Client

While FTP server support is still less than perfect in IPF, FTP client support has been working well since 3.3.3. As with FTP servers, there are two types of ftp client transfers: passive and active.

The simplest type of client transfer from the firewall's standpoint is the passive transfer. Assuming you're keeping state on all outbound tcp sessions, passive transfers will work already. If you're not doing this already, please consider the following:

```
pass out proto tcp all keep state
```

The second type of client transfer, active, is a bit more troublesome, but nonetheless a solved problem. Active transfers cause the server to open up a second connection back to the client for data to flow through. This is normally a problem when there's a firewall in the middle, stopping outside connections from coming back in. To solve this, ipfilter includes an `ipnat` proxy which temporarily opens up a hole in the firewall just for the FTP server to get back to the client. Even if you're not using `ipnat` to do nat, the proxy is still effective. The following rules is the bare minimum to add to the `ipnat` configuration file (`ep0` should be the interface name of the outbound network connection):

```
map ep0 0/0 -> 0/32 proxy port 21 ftp/tcp
```

For more details on ipfilter's internal proxies, see section 3.6. Additionally, IPF's ftp proxy does work the "wrong way", and can be used to support a natted FTP server, but you *really* don't want to do this for security reasons. Really. This is a huge security hole. See http://www.false.net/ipfilter/2001_11/0273.html for reasons why you should forget you ever thought about this.

8.3. Assorted Kernel Variables

There are some useful kernel tunes that either need to be set for ipf to function, or are just generally handy to know about for building firewalls. The first major one you must set is to enable IP Forwarding, otherwise ipf will do very little, as the underlying ip stack won't actually route packets.

IP Forwarding:

openbsd:
net.inet.ip.forwarding=1

freebsd:
net.inet.ip.forwarding=1

netbsd:
net.inet.ip.forwarding=1

solaris:
ndd -set /dev/ip ip_forwarding 1

Ephemeral Port Adjustment:

openbsd:
net.inet.ip.portfirst = 25000

freebsd:
net.inet.ip.portrange.first = 25000 net.inet.ip.portrange.last = 49151

netbsd:

net.inet.ip.anonportmin = 25000 net.inet.ip.anonportmax = 49151

solaris:

ndd -set /dev/tcp tcp_smallest_anon_port 25000
ndd -set /dev/tcp tcp_largest_anon_port 65535

Other Useful Values:

openbsd:

net.inet.ip.sourceroute = 0
net.inet.ip.directed-broadcast = 0

freebsd:

net.inet.ip.sourceroute=0
net.ip.accept_sourceroute=0

netbsd:

net.inet.ip.allowsrcrt=0
net.inet.ip.forwsrcrt=0
net.inet.ip.directed-broadcast=0
net.inet.ip.redirect=0

solaris:

ndd -set /dev/ip ip_forward_directed_broadcasts 0
ndd -set /dev/ip ip_forward_src_routed 0
ndd -set /dev/ip ip_respond_to_echo_broadcast 0

In addition, freebsd has some ipf specific sysctl variables.

```
net.inet.ipf.fr_flags: 0
net.inet.ipf.fr_pass: 514
net.inet.ipf.fr_active: 0
net.inet.ipf.fr_tcpidletimeout: 864000
net.inet.ipf.fr_tcpclosewait: 60
net.inet.ipf.fr_tcplastack: 20
net.inet.ipf.fr_tcptimeout: 120
net.inet.ipf.fr_tcpclosed: 1
net.inet.ipf.fr_udptimeout: 120
net.inet.ipf.fr_icmptimeout: 120
net.inet.ipf.fr_defnatage: 1200
net.inet.ipf.fr_ipfrttl: 120
net.inet.ipf.ipl_unreach: 13
net.inet.ipf.ipl_inited: 1
net.inet.ipf.fr_authsize: 32
net.inet.ipf.fr_authused: 0
net.inet.ipf.fr_defaultauthage: 600
```

9. Fun with ipf!

This section doesn't necessarily teach you anything new about ipf, but it may raise an issue or two that you haven't yet thought up on your own, or tickle your brain in a way that you invent something interesting that we haven't thought of.

9.1. Localhost Filtering

A long time ago at a university far, far away, Wietse Venema created the tcp-wrapper package, and ever since, it's been used to add a layer of protection to network services all over the world. This is good. But, tcp-wrappers have flaws. For starters, they only protect TCP services, as the name suggests. Also, unless you run your service from inetd, or you have specifically compiled it with libwrap and the appropriate hooks, your service isn't protected. This leaves gigantic holes in your host security. We can plug these up by using ipf on the local host. For example, my laptop often gets plugged into or dialed into networks that I don't specifically trust, and so, I use the following rule set:

```
pass in quick on lo0 all
pass out quick on lo0 all

block in log all
block out all

pass in quick proto tcp from any to any port = 113 flags S keep state
pass in quick proto tcp from any to any port = 22 flags S keep state
pass in quick proto tcp from any port = 20 to any port 39999 >< 45000 flags S keep state

pass out quick proto icmp from any to any keep state
pass out quick proto tcp/udp from any to any keep state keep frags
```

It's been like that for quite a while, and I haven't suffered any pain or anguish as a result of having ipf loaded up all the time. If I wanted to tighten it up more, I could switch to using the NAT ftp proxy and I could add in some rules to prevent spoofing. But even as it stands now, this box is far more restrictive about what it presents to the local network and beyond than the typical host does. This is a good thing if you happen to run a machine that allows a lot of users on it, and you want to make sure one of them doesn't happen to start up a service they wern't supposed to. It won't stop a malicious hacker with root access from adjusting your ipf rules and starting a service anyway, but it will keep the "honest" folks honest, and your weird services safe, cozy and warm even on a malicious LAN. A big win, in my opinion. Using local host filtering in addition to a somewhat less-restrictive "main firewall" machine can solve many performance issues as well as *political* nightmares like "Why doesn't ICQ work?" and "Why can't I put a web server on my own workstation! It's MY WORKSTATION!!" Another very big win. Who says you can't have security and convienence at the same time?

9.2. What Firewall? Transparent filtering.

One major concern in setting up a firewall is the integrity of the firewall itself. Can somebody break into your firewall, thereby subverting its ruleset? This is a common problem administrators must face, particularly when they're using firewall solutions on top of their Unix/NT machines. Some use it as an arguement for blackbox hardware solutions, under the flawed notion that inherent obscurity of their closed system increases their security. We have a better way.

Many network admins are familiar with the common ethernet bridge. This is a device that connects two separate ethernet segments to make them one. An ethernet bridge is typically used to connect separate buildings, switch network speeds, and extend maximum wire lengths. Hubs and switches are common bridges, sometimes they're just 2 ported devices called repeaters. Recent versions of Linux, OpenBSD, NetBSD, and FreeBSD include code to convert $1000 PCs into $10 bridges, too! What all bridges tend to have in common is that though they sit in the middle of a connection between two machines, the two machines don't know the bridge is there. Enter ipfilter and OpenBSD.

Ethernet bridging takes place at Layer2 on the ISO stack. IP takes place on Layer3. IP Filter in primarily concerned with Layer3, but dabbles in Layer2 by working with interfaces. By mixing IP filter with OpenBSD's bridge device, we can create a firewall that is both invisible and unreachable. The system needs no IP address, it doesn't even need to reveal its ethernet address. The only telltale sign that the filter might be there is that latency is somewhat higher than a piece of cat5 would normally make it, and that packets don't seem to make it to their final destination.

The setup for this sort of ruleset is surprisingly simple, too. In OpenBSD, the first bridge device is named `bridge0`. Say we have two ethernet cards in our machine as well, `xl0` and `xl1`. To turn this machine into a bridge, all one need do is enter the following three commands:

```
brconfig bridge0 add xl0 add xl1 up
ifconfig xl0 up
ifconfig xl1 up
```

At ths point, all traffic ariving on `xl0` is sent out `xl1` and all traffic on `xl1` is sent out `xl0`. You'll note that neither interface has been assigned an IP address, nor do we need assign one. All things considered, it's likely best we not add one at all.

Rulesets behave essentially the as the always have. Though there is a `bridge0` interface, we don't filter based on it. Rules continue to be based upon the particular interface we're using, making it important which network cable is plugged into which network card in the back of the machine. Let's start with some basic filtering to illistrate what's happened. Assume the network used to look like this:

```
20.20.20.1 <---------------------------------> 20.20.20.0/24 network hub
```

That is, we have a router at 20.20.20.1 connected to the 20.20.20.0/24 network. All packets from the 20.20.20.0/24 network go through 20.20.20.1 to get to the outside world and vice versa. Now we add the Ipf Bridge:

```
20.20.20.1 <-------/xl0 IpfBridge xl1/-------> 20.20.20.0/24 network hub
```

We also have the following ruleset loaded on the IpfBridge host:

```
pass in  quick  all
pass out quick  all
```

With this ruleset loaded, the network is functionally identical. As far as the 20.20.20.1 router is concerned, and as far as the 20.20.20.0/24 hosts are concerned, the two network diagrams are identical. Now let's change the ruleset some:

```
block in quick on xl0 proto icmp
pass  in  quick all
pass  out quick all
```

Still, 20.20.20.1 and 20.20.20.0/24 think the network is identical, but if 20.20.20.1 attempts to ping 20.20.20.2, it will never get a reply. What's more, 20.20.20.2 won't even get the packet in the first place. IPfilter will intercept the packet before it even gets to the other end of the virtual wire. We can put a bridged filter anywhere. Using this method we can shrink the network trust circle down an individual host level (given enough ethernet cards:-)

Blocking icmp from the world seems kind of silly, especially if you're a sysadmin and like pinging the world, to traceroute, or to resize your MTU. Let's construct a better ruleset and take advantage of the original key feature of ipf: stateful inspection.

```
pass  in quick on xl1 proto tcp  keep state
pass  in quick on xl1 proto udp  keep state
pass  in quick on xl1 proto icmp keep state
block in quick on xl0
```

In this situation, the 20.20.20.0/24 network (perhaps more aptly called the `xl1` network) can now reach the outside world, but the outside world can't reach it, and it can't figure out why, either. The router is accessible, the hosts are active, but the outside world just can't get in. Even if the router itself were compromised, the firewall would still be active and successful.

So far, we've been filtering by interface and protocol only. Even though bridging is concerned layer2, we can still discriminate based on IP address. Normally we have a few services running, so our ruleset may look like this:

```
pass  in quick on xl1 proto tcp  keep state
pass  in quick on xl1 proto udp  keep state
pass  in quick on xl1 proto icmp keep state
block in quick on xl1 # nuh-uh, we're only passing tcp/udp/icmp sir.
pass  in quick on xl0 proto udp from any to 20.20.20.2/32 port=53 keep state
pass  in quick on xl0 proto tcp from any to 20.20.20.2/32 port=53 flags S keep state
pass  in quick on xl0 proto tcp from any to 20.20.20.3/32 port=25 flags S keep state
pass  in quick on xl0 proto tcp from any to 20.20.20.7/32 port=80 flags S keep state
block in quick on xl0
```

Now we have a network where 20.20.20.2 is a zone serving name server, 20.20.20.3 is an incoming mail server, and 20.20.20.7 is a web server.

Bridged IP Filter is not yet perfect, we must confess.

First, You'll note that all the rules are setup using the `in` direction instead of a combination of `in` and `out`. This is because the `out` direction is presently unimplemented with bridging in OpenBSD. This was originally done to prevent vast performance drops using multiple interfaces. Work has been done in speeding it up, but it remains unimplemented. If you really want this feature, you might try your hand at working on the code or asking the OpenBSD people how you can help.

Second, using IP Filter with bridging makes the use of IPF's NAT features inadvisable, if not downright dangerous. The first problem is that it would give away that there's a filtering bridge. The second problem would be that the bridge has no IP address to masquerade with, which will most assuredly lead to confusion and perhaps a kernel panic to boot. You can, of course, put an IP address on the outbound interface to make NAT work, but part of the glee of bridging is thus diminished.

9.2.1. Using Transparent Filtering to Fix Network Design Mistakes

Many organizations started using IP well before they thought a firewall or a subnet would be a good idea. Now they have class-C sized networks or larger that include all their servers, their workstations, their routers, coffee makers, everything. The horror! Renumbering with proper subnets, trust levels, filters, and so are in both time consuming and expensive. The expense in hardware and man hours alone is enough to make most organizations unwilling to really solve the problem, not to mention the downtime involved. The typical problem network looks like this:

```
20.20.20.1   router               20.20.20.6   unix server
20.20.20.2   unix server          20.20.20.7   nt workstation
20.20.20.3   unix server          20.20.20.8   nt server
20.20.20.4   win98 workstation    20.20.20.9   unix workstation
20.20.20.5   intelligent switch   20.20.20.10  win95 workstation
```

Only it's about 20 times larger and messier and frequently undocumented. Ideally, you'd have all the trusting servers in one subnet, all the work- stations in another, and the network switches in a third. Then the router would filter packets between the subnets, giving the workstations limited access to the servers, nothing access to the switches, and only the sysadmin's workstation access to the coffee pot. I've never seen a class-C sized network with such coherence. IP Filter can help.

To start with, we're going to separate the router, the workstations, and the servers. To do this we're going to need 2 hubs (or switches) which we probably already have, and an IPF machine with 3 ethernet cards. We're going to put all the servers on one hub and all the workstations on the other. Normally we'd then connect the hubs to each other, then to the router. Instead, we're going to plug the router into IPF's `xl0` interface, the servers into IPF's `xl1` interface, and the workstations into IPF's `xl2` interface. Our network diagram looks something like this:

```
                                              | 20.20.20.2  unix server
router (20.20.20.1)              _____________| 20.20.20.3  unix server
 |                               /            | 20.20.20.6  unix server
 |                              /xl1          | 20.20.20.7  nt server
 ------------/xl0 IPF Bridge <
                                 xl2         | 20.20.20.4  win98 workstation
                                 ____________| 20.20.20.8  nt workstation
                                              | 20.20.20.9  unix workstation
                                              | 20.20.20.10 win95 workstation
```

Where once there was nothing but interconnecting wires, now there's a filtering bridge that not a single host needs to be modified to take advantage of. Presumably we've already enabled bridging so the network is behaving perfectly normally. Further, we're starting off with a ruleset much like our last ruleset:

```
pass  in quick on xl0 proto udp from any to 20.20.20.2/32 port=53 keep state
pass  in quick on xl0 proto tcp from any to 20.20.20.2/32 port=53 flags S keep state
pass  in quick on xl0 proto tcp from any to 20.20.20.3/32 port=25 flags S keep state
pass  in quick on xl0 proto tcp from any to 20.20.20.7/32 port=80 flags S keep state
block in quick on xl0
pass  in quick on xl1 proto tcp  keep state
pass  in quick on xl1 proto udp  keep state
pass  in quick on xl1 proto icmp keep state
block in quick on xl1 # nuh-uh, we're only passing tcp/udp/icmp sir.
pass  in quick on xl2 proto tcp  keep state
pass  in quick on xl2 proto udp  keep state
pass  in quick on xl2 proto icmp keep state
block in quick on xl2 # nuh-uh, we're only passing tcp/udp/icmp sir.
```

Once again, traffic coming from the router is restricted to DNS, SMTP, and HTTP. At the moment, the servers and the workstations can exchange traffic freely. Depending on what kind of organization you are, there might be something about this network dynamic you don't like. Perhaps you don't want your workstations getting access to your servers at all? Take the `xl2` ruleset of:

```
pass  in quick on xl2 proto tcp  keep state
pass  in quick on xl2 proto udp  keep state
pass  in quick on xl2 proto icmp keep state
block in quick on xl2 # nuh-uh, we're only passing tcp/udp/icmp sir.
```

And change it to:

```
block in quick on xl2 from any to 20.20.20.0/24
pass  in quick on xl2 proto tcp  keep state
pass  in quick on xl2 proto udp  keep state
pass  in quick on xl2 proto icmp keep state
block in quick on xl2 # nuh-uh, we're only passing tcp/udp/icmp sir.
```

Perhaps you want them to just get to the servers to get and send their mail with IMAP? Easily done:

```
pass  in quick on xl2 proto tcp from any to 20.20.20.3/32 port=25
pass  in quick on xl2 proto tcp from any to 20.20.20.3/32 port=143
block in quick on xl2 from any to 20.20.20.0/24
pass  in quick on xl2 proto tcp  keep state
pass  in quick on xl2 proto udp  keep state
pass  in quick on xl2 proto icmp keep state
block in quick on xl2 # nuh-uh, we're only passing tcp/udp/icmp sir.
```

Now your workstations and servers are protected from the outside world, and the servers are protected from your workstations.

Perhaps the opposite is true, maybe you want your workstations to be able to get to the servers, but not the outside world. After all, the next generation of exploits is breaking the clients, not the servers. In this case, you'd change the `xl2` rules to look more like this:

```
pass  in quick on xl2 from any to 20.20.20.0/24
block in quick on xl2
```

Now the servers have free reign, but the clients can only connect to the servers. We might want to batten down the hatches on the servers, too:

```
pass  in quick on xl1 from any to 20.20.20.0/24
block in quick on xl1
```

With the combination of these two, the clients and servers can talk to each other, but neither can access the outside world (though the outside world can get to the few services from earlier). The whole ruleset would look something like this:

```
pass  in quick on xl0 proto udp from any to 20.20.20.2/32 port=53 keep state
pass  in quick on xl0 proto tcp from any to 20.20.20.2/32 port=53 flags S keep state
pass  in quick on xl0 proto tcp from any to 20.20.20.3/32 port=25 flags S keep state
pass  in quick on xl0 proto tcp from any to 20.20.20.7/32 port=80 flags S keep state
block in quick on xl0
pass  in quick on xl1 from any to 20.20.20.0/24
block in quick on xl1
pass  in quick on xl2 from any to 20.20.20.0/24
block in quick on xl2
```

So remember, when your network is a mess of twisty IP addresses and machine classes, transparent filtered bridges can solve a problem that would otherwise be lived with and perhaps someday exploited.

9.3. Drop-Safe Logging With dup-to and to.

Until now, we've been using the filter to drop packets. Instead of dropping them, let's consider passing them on to another system that can do something useful with this information beyond the logging we can perform with ipmon. Our firewall system, be it a bridge or a router, can have as many interfaces as we can cram into the system. We can use this information to create a "drop-safe" for our packets. A good example of a use for this would be to implement an intrusion detection network. For starters, it might be desirable to hide the presence of our intrusion detection systems from our real network so that we can keep them from being detected.

Before we get started, there are some operational characteristics that we need to make note of. If we are only going to deal with blocked packets, we can use either the `to` keyword or the `fastroute` keyword. (We'll cover the differences between these two later) If we're going to pass the packets like we normally would, we need to make a copy of the packet for our drop-safe log with the `dup-to` keyword.

9.3.1. The dup-to Method

If, for example, we wanted to send a copy of everything going out the `xl3` interface off to our drop-safe network on `ed0`, we would use this rule in our filter list:

```
pass out on xl3 dup-to ed0 from any to any
```

You might also have a need to send the packet directly to a specific IP address on your drop-safe network instead of just making a copy of the packet out there and hoping for the best. To do this, we modify our rule slightly:

```
pass out on xl3 dup-to ed0:192.168.254.2 from any to any
```

But be warned that this method will alter the copied packet's destination address, and may thus destroy the usefulness of the log. For this reason, we recommend only using the known address method of logging when you can be certain that the address that you're logging to corresponds in some way to what you're logging for (e.g.: don't use "192.168.254.2" for logging for both your web server and your mail server, since you'll have a hard time later trying to figure out which system was the target of a specific set of packets.)

This technique can be used quite effectively if you treat an *IP Address* on your drop-safe network in much the same way that you would treat a *Multicast Group* on the real internet. (e.g.: "192.168.254.2" could be the channel for your http traffic analysis system, "23.23.23.23" could be your channel for telnet sessions, and so on.) You don't even need to actually have this address set as an address or alias on any of your analysis systems. Normally, your ipfilter machine would need to ARP for the new destination address (using `dup-to ed0:192.168.254.2` style, of course) but we can avoid that issue by creating a static arp entry for this *"channel"* on our ipfilter system.

In general, though, `dup-to ed0` is all that is required to get a new copy of the packet over to our drop-safe network for logging and examination.

9.3.2. The to Method

The `dup-to` method does have an immediate drawback, though. Since it has to make a copy of the packet and optionally modify it for its new destination, it's going to take a while to complete all this work and be ready to deal with the next packet coming in to the ipfilter system.

If we don't care about passing the packet to its normal destination and we were going to block it anyway, we can just use the `to` keyword to push this packet past the normal routing table and force it to go out a different interface than it would normally go out.

```
block in quick on xl0 to ed0 proto tcp from any to any port < 1024
```

we use `block quick` for `to` interface routing, because like `fastroute`, the `to` interface code will generate two packet paths through ipfilter when used with `pass`, and likely cause your system to panic.

10. Bogus Network Filtering, the ultimate in current anti-spoofing technology.

We've spent a little bit of time tracking down the current vast tracts of IP address space that have been reserved by the IANA for various reasons, or are otherwise not currently in use at the time this document was written. Since none of these address ranges should be in use currently, there should be no legitimate reason to ever see them as a source address, or to send them traffic as a destination address, right? Right!

So without further ado, the complete list of bogus networks:

```
#
# s/OUTSIDE/outside-interface (eg: fxp0)
# s/MYNET/network-cidr-address (eg: 1.2.3.0/24)
#
block in on OUTSIDE all
block in quick on OUTSIDE from 0.0.0.0/7 to any
```

```
block in quick on OUTSIDE from 2.0.0.0/8 to any
block in quick on OUTSIDE from 5.0.0.0/8 to any
block in quick on OUTSIDE from 10.0.0.0/8 to any
block in quick on OUTSIDE from 23.0.0.0/8 to any
block in quick on OUTSIDE from 27.0.0.0/8 to any
block in quick on OUTSIDE from 31.0.0.0/8 to any
block in quick on OUTSIDE from 70.0.0.0/7 to any
block in quick on OUTSIDE from 72.0.0.0/5 to any
block in quick on OUTSIDE from 83.0.0.0/8 to any
block in quick on OUTSIDE from 84.0.0.0/6 to any
block in quick on OUTSIDE from 88.0.0.0/5 to any
block in quick on OUTSIDE from 96.0.0.0/3 to any
block in quick on OUTSIDE from 127.0.0.0/8 to any
block in quick on OUTSIDE from 128.0.0.0/16 to any
block in quick on OUTSIDE from 128.66.0.0/16 to any
block in quick on OUTSIDE from 169.254.0.0/16 to any
block in quick on OUTSIDE from 172.16.0.0/12 to any
block in quick on OUTSIDE from 191.255.0.0/16 to any
block in quick on OUTSIDE from 192.0.0.0/19 to any
block in quick on OUTSIDE from 192.0.48.0/20 to any
block in quick on OUTSIDE from 192.0.64.0/18 to any
block in quick on OUTSIDE from 192.0.128.0/17 to any
block in quick on OUTSIDE from 192.168.0.0/16 to any
block in quick on OUTSIDE from 197.0.0.0/8 to any
block in quick on OUTSIDE from 201.0.0.0/8 to any
block in quick on OUTSIDE from 204.152.64.0/23 to any
block in quick on OUTSIDE from 219.0.0.0/8 to any
block in quick on OUTSIDE from 220.0.0.0/6 to any
block in quick on OUTSIDE from 224.0.0.0/3 to any
block in quick on OUTSIDE from MYNET to any
# Your pass rules come here...

block out on OUTSIDE all
block out quick on OUTSIDE from !MYNET to any
block out quick on OUTSIDE from MYNET to 0.0.0.0/7
block out quick on OUTSIDE from MYNET to 2.0.0.0/8
block out quick on OUTSIDE from MYNET to 5.0.0.0/8
block out quick on OUTSIDE from MYNET to 10.0.0.0/8
block out quick on OUTSIDE from MYNET to 23.0.0.0/8
block out quick on OUTSIDE from MYNET to 27.0.0.0/8
block out quick on OUTSIDE from MYNET to 31.0.0.0/8
block out quick on OUTSIDE from MYNET to 70.0.0.0/7
block out quick on OUTSIDE from MYNET to 72.0.0.0/5
block out quick on OUTSIDE from MYNET to 83.0.0.0/8
block out quick on OUTSIDE from MYNET to 84.0.0.0/6
block out quick on OUTSIDE from MYNET to 88.0.0.0/5
block out quick on OUTSIDE from MYNET to 96.0.0.0/3
block out quick on OUTSIDE from MYNET to 127.0.0.0/8
block out quick on OUTSIDE from MYNET to 128.0.0.0/16
block out quick on OUTSIDE from MYNET to 128.66.0.0/16
block out quick on OUTSIDE from MYNET to 169.254.0.0/16
block out quick on OUTSIDE from MYNET to 172.16.0.0/12
block out quick on OUTSIDE from MYNET to 191.255.0.0/16
block out quick on OUTSIDE from MYNET to 192.0.0.0/19
block out quick on OUTSIDE from MYNET to 192.0.48.0/20
block out quick on OUTSIDE from MYNET to 192.0.64.0/18
block out quick on OUTSIDE from MYNET to 192.0.128.0/17
block out quick on OUTSIDE from MYNET to 192.168.0.0/16
block out quick on OUTSIDE from MYNET to 197.0.0.0/8
block out quick on OUTSIDE from MYNET to 201.0.0.0/8
block out quick on OUTSIDE from MYNET to 204.152.64.0/23
block out quick on OUTSIDE from MYNET to 219.0.0.0/8
block out quick on OUTSIDE from MYNET to 220.0.0.0/6
block out quick on OUTSIDE from MYNET to 224.0.0.0/3
# Your pass rules come here...
```

If you're going to use these, we suggest that you become familiar with whois.arin.net and keep an occasional eye on these, as the IANA isn't going to notify you when they allocate one of these to a new corporation or something. You have been warned.