Building a firewall with FreeBSD **(last update: 03/18/2004)**

In this document I'm going to try to explain how to build a pretty full-fledged firewall with **stateful packet filtering, NAT, optional PPPoE client support and an optional DMZ port**. We'll do this by using ipfilter+ipnat, not the ipfw+natd combination. I'll explain the reasons for this choice in a minute.

I have been using a FreeBSD machine as a means to share my ADSL internet connection with several other computers on my small LAN for about a year, and I have never had any problems with it so far – no crashes, no security problems, no performance worries. However, if you don't feel comfortable with setting up a firewall on your own, for all means go and buy one of those built-to-run firewall appliances like SonicWall, ZyWALL, CheckPoint, Watchguard, whatever.

I assume you have enough knowledge about FreeBSD to install and configure the basic stuff, including networking. Some firewall knowledge can't hurt either – that way you'll actually understand why the ruleset has to be the way it is, etc. I recommend reading the excellent book *Building Internet Firewalls, 2nd Edition* by *Elizabeth D. Zwicky, Simon Cooper and D. Brent Chapman* (ISBN 1-56592-871-7).

Note that I won't guarantee that building a firewall using this guide will make your network absolutely secure. There are many other factors involved in security, and absolute security does not exist anyway. I assume that a firewall built with this guide should be pretty secure, however if, for example, you just open a single port on the firewall, that may change from pretty secure to totally insecure if something's wrong with the application/OS/machine listening on that port behind the firewall. There may be bugs in ipfilter or FreeBSD, too – who knows?

So, by using this guide to build a firewall, you agree to take all liability for your actions. Feel free to read my standard disclaimer, which applies to this guide, too.

If you have any suggestions, success reports or whatever, please let me know at <mk@neon1.net>. You can also email me with questions, but please be aware that I can't answer all newbie's questions – search at http://groups.google.com/ first; I can almost always find the answer to my questions there.

Finally, I'm sorry for any spelling/grammar mistakes – English isn't my native language (if you want to proof-read this document – go ahead! :).

Table of contents

Revision history

10/02/2002

- initial release

Introduction

Why ipfilter+ipnat and not ipfw+natd?
As I mentioned earlier on, we're going to use *ipfilter+ipnat* instead of the (more common?) *ipfw+natd* combination. I've been using ipfw+natd for some time, and ipfw per-se is a very nice firewall. However, it lacks a well-integrated NAT implementation. The concept of divert sockets along with a userland program (natd) doing the actual NAT is OK, but it does not fit well in the ipfw architecture. There's one problem that most of the people trying to do some advanced filtering with ipfw+natd will run into: incoming packets are not handled in the same way as outgoing packets. I mean, the divert rule which sends packets to natd usually comes before all other filtering rules. Now here's what happens: incoming packets are translated before being processed by the firewall rules. This means that their destination address will be the actual, private/internal address of the destination host. Very well. However, since outgoing packets get translated before rule processing, too, this means that their source address will always be equal to the address of the external interface of the firewall. This makes it impossible to tell if outgoing packets originate from a host behind the firewall or from the firewall itself when they are passing through the WAN interface, and breaks the ipfw keep-state feature, since it will not see the same address pair for incoming and outgoing packets (remote address <--> firewall address vs. remote address <--> internal host address).

The "solution" is either to not use keep-state rules or to change the ruleset to do NAT for incoming packets before the firewall rules and for outgoing packets after the firewall rules. That, however, means that you're going to use a lot of skipto rules which render the ruleset rather messy and hard-to-understand.

Loophole for PPP connections
There is a loophole, however – if your internet connection requires using userland ppp (for example, with PPPoE connections), you can enable NAT in ppp. That's prettty much the same thing as natd since it's based on the same library (libalias), but because NAT is done in ppp without requiring any divert rules, the packets will always get NATed at the right moment (before firewall processing for incoming packets, after processing for outgoing packets).

Quest for the "perfect" solution
I'm a little perfectionist, so since I couldn't find a "clean" solution with ipfw+natd, I decided to give the ominous ipfilter a try. It couldn't hurt, since ipfilter is a standard in other sytems as well (for example, OpenBSD, where it's called *pf* and has a compatible rule language). The result? I immediately fell in love with it, and there's only one feature I'm missing: dummynet support for traffic shaping. However, you can still use ipfw just for traffic shaping and let ipfilter do the actual filtering – you just have to pay attention to the order in which packets pass through ipfw and ipfilter.

Last but not least, if you're looking for a nice little appliance to be used as a firewall, check out the net4501 from Soekris Engineering and my guide on getting FreeBSD to run on it. It's an SBC (Single Board Computer) with 3 x 10/100 LAN interfaces, a 486/133 MHz CPU and 64 MB SDRAM – more than enough to do firewalling for internet connections up to about 10 Mbps. I'm running ipfilter on such a machine, and so far it's working flawlessly.

---

1. Installing FreeBSD

First of all, you need a standard FreeBSD installation on a machine equipped with at least two network interfaces (three if you want a DMZ). I will not cover this topic in detail, because I assume you know FreeBSD to some degree (or are willing to learn :). A minimal installation will suffice. This guide is based on FreeBSD 4.6, although it should work with earlier/future versions, too.

---

2. Enabling ipfilter

ipfilter is not enabled by default in FreeBSD (ipfw isn't, either), so the first step we have to take is to enable it. Make sure you're sitting at a local console of the machine – never tinker with the firewall via a remote connection, as it's very easy to lock yourself out!

Edit `/etc/rc.conf` and add the following lines:

```
ipfilter_enable="YES"
ipnat_enable="YES"
```

rc.network should automatically load ipfilter as a module if it's not compiled into the kernel. If you want to compile it into the kernel, here are the options you have to add to your kernel configuration file:

```
options IPFILTER
options IPFILTER_LOG
options IPFILTER_DEFAULT_BLOCK
```

Again, remember that this kernel configuration will block all packets by default unless you provide some filter rules (see chapter 5) to permit the traffic you want. Reboot, and ipfilter should be enabled and ready for configuration.

---

3. Setting up PPPoE (optional)

Before we do any firewall/NAT configuration, we should get PPPoE (or another kind of ppp connection, for that matter) running, if necessary. This step is only required if your internet connection requires using some flavour of PPP. In Switzerland, where I live, all ADSL connections require using PPPoE (PPP over Ethernet), so here's how to set it up:

edit `/etc/ppp/ppp.conf` and put the following lines into it:

```
default:
   set log Phase tun command
   set ifaddr 10.0.0.1/0 10.0.0.2/0

myisp:
   set device PPPoE:sis1
   set MRU 1492
   set MTU 1492
   set lqrperiod 20
   enable lqr
   set authname username
```

```
   set authkey password
   set dial
   set login
   add default HISADDR
```

Ignore the 10.0.0.x addresses in the third line; they're just placeholders for the real, dynamically assigned address. Replace the interface name (sis1 in this example) with the name of the interface to which your xDSL modem is attached. Insert the username/password required for PPPoE in the appropriate places, too. If you want to, change "myisp" to reflect the name of your ISP. If you have problems logging on, try removing the "`enable lqr`" and "`set lqrperiod 20`" lines. Some providers do not support PPP LQR, which serves to help ppp find out if there's a problem with the link. If it doesn't get any answer from the remote side after 5 * lqrperiod seconds, it will decide that the ppp connection has died and try to set it up again immediately. This is very important for ADSL since there's no way for ppp to determine when the ADSL link goes down without prior notice.

MTU
The PPPoE header which gets prepended to each packet that goes over the PPP connection has a length of 8 bytes. This means that the IP packets cannot be longer than 1492 bytes, otherwise they will have to be fragmented or thrown away. Since your LAN hosts are most likely connected via Ethernet, they'll assume that they can send packets of up to 1500 bytes length. Fragmenting them just because of 8 lousy bytes is pretty stupid (as you'll have to send the whole IP header again), and some internet hosts do not even accept IP fragments because they can be used for DoS (Denial of Service) attacks (it takes quite a bit of processor time to reassemble IP fragments).
ppp has an option that enables it to behave like most PPPoE capable xDSL routers: they can silently adjust the MSS (Maximum Segment Size, which is the maximum acceptable size of an IP packet minus the length of the IP headers) of outgoing TCP SYN packets. By specifying "`set MTU 1492`" you can enable that function and circumvent the problem without having to change the MTU on your LAN hosts.

Now, since the tunnel interface brought up by ppp does not exist at the time ipfilter is loaded, we have to tell ipfilter about the new interface each time the ppp connection goes up (as the IP address may have changed). So create/edit the file `/etc/ppp/ppp.linkup` and add the following lines to it:

```
myisp:
   !bg /sbin/ipf -y
```

Of course, change "myisp" if necessary. This means that ppp will invoke `ipf -y` each time the ppp connection goes up, so ipfilter gets a chance to refresh its internal interface table.

Add the following lines to `/etc/rc.conf` to automatically start ppp on boot:

```
ppp_enable="YES"
ppp_mode="ddial"
ppp_profile="myisp"
```

"myisp" has to match the label you chose in your ppp.conf. You can now start ppp manually:

```
ppp -quiet -ddial myisp
```

Now check the output from `ifconfig`. You should see a new interface, usually tun0, with the IP address that got assigned to you by your provider (this may take a few seconds after ppp is started), and you should be able to connect to the internet from the firewall machine (try using ftp, for example).

If not, something's obviously wrong. Check if everything's correct (wiring, username, password, etc.), and if that doesn't help, have a look at the PPPoE chapter of the FreeBSD handbook:

http://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/pppoe.html

---

4. Setting up NAT

We'll set up NAT first as it's easier to configure. The rules for ipnat are specified in a text file, `/etc/ipnat.rules`. So create that file and put the following lines into it:

```
map tun0 0/0 -> 0/32 proxy port ftp ftp/tcp
map tun0 0/0 -> 0/32 portmap tcp/udp auto
map tun0 0/0 -> 0/32
```

If you're not using PPP(oE), you'll have to change tun0 to reflect your external interface name (that is, the one that's connected to the internet). ipnat rules are processed on a first-match basis; the first rule will catch connections on the ftp port (21) and pass them to the ftp proxy module. That module will take care to translate the IP addresses in the PASV/PORT commands and also automatically add dynamic rules to the firewall for active FTP data connections (or incoming passive FTP data connections if you run an FTP server behind your firewall). Very beautiful. There are proxies for other hard-to-firewall protocols with dynamically allocated ports like H.323 or RPC as well.

The second rule will catch TCP and UDP connections, and the third rule is for everything else (IP protocols like ICMP, GRE (PPTP), ESP (IPsec) and so on). We're now ready to load the ipnat rules:

```
ipnat -F -f /etc/ipnat.rules
```

You can now try accessing the internet from a machine behind the firewall – but be aware that there's no firewall security yet! You should only have to set the default gateway of the machines on the LAN to the internal (LAN) IP address of the firewall, and make sure a name (DNS) server is configured on them (you can use your ISPs DNS server).

---

5. Setting up the firewall

Now it's time to put together the firewall ruleset. We'll use dynamic rules (stateful filtering) and default to denying everything which is not explicitly allowed. You should familiarize yourself with the way ipfilter rules are processed – it's a bit special in the way that the last rule to match a packet will be applied (unless you use the keyword 'quick', which is what we're going to do most of the time). Read the manpage for the ipf rule syntax:

```
man 5 ipf
```

The ruleset is usually stored in `/etc/ipf.rules`.

The state table
When ipfilter processes a packet, it will first take a look at the state table to see if there's an entry that matches the source/destination IP address (and port, in case of TCP/UDP) combination. If there is, the packet will be passed without checking the firewall rules. This means we'll only have to deal with packets that mark the beginning of a connection – in case of TCP, that's the SYN packet – and we'll use "keep state" with all of our pass rules to have ipfilter add an entry to the state table.

So here's a ruleset that (almost) only permits packets belonging to connections that originated from a host on the LAN. In this example, the external interface is tun0 and the internal interface is sis0. The

LAN hosts are on subnet 192.168.0.0/24 and the firewall has the IP address 192.168.0.254 on its LAN interface. Change the interface names/addresses to suit your needs. First, a few rules that apply to all packets:

```
# block short packets
block in quick all with short

# block packets with IP options
block in quick all with ipopts

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

The first rule will block "short" packets – that is, packets that don't contain enough information to be assigned with a particular connection. Those are usually maliciously generated IP fragments and should not happen with normal, legitimate usage, so the best policy is to block them. Packets with IP options are blocked, too, because they're not normally useful but can be used in attacks as well (source routing, etc.). All traffic is permitted on the loopback interface (lo0).

Now for the head rules:

```
#------------------------------------------------------------------------
-
# group head 100/150 - LAN interface
#------------------------------------------------------------------------
-
block in quick on sis0 all head 100
block out quick on sis0 all head 150

#------------------------------------------------------------------------
-
# group head 200/250 - WAN interface
#------------------------------------------------------------------------
-
block in log quick on tun0 all head 200
block out quick on tun0 all head 250
```

When ipfilter encounters a head rule that matches the given packet, it will continue by processing all rules which have the same group number. If none of these group rules matches, the action that is specified by the head rule will be taken. Our head rules all have "block quick", so everything that isn't explicitly allowed by a group rule gets blocked. Head rules make it possible to process rules by interface and direction, so we can specify separate rulesets for incoming and outgoing packets on the LAN and the WAN (and later DMZ) interfaces.
Note that the head rule for incoming packets on the WAN interface has 'log' specified – this means that we can run ipmon and see which incoming packets on the WAN interface were denied (see chapter 7).

Incoming packets on the LAN interface
Let's take a look at the ruleset for incoming packets on the LAN interface first. We said we'll only permit outgoing connections from LAN -> WAN and LAN -> DMZ, so outgoing packets for LAN hosts are incoming packets on the LAN interface for the firewall machine:

```
#------------------------------------------------------------------------
-
# incoming traffic on LAN interface - group 100
#------------------------------------------------------------------------
-
```

```
pass in quick proto tcp from 192.168.0.0/24 to any flags S/SAFR keep state
group 100
pass in quick proto udp from 192.168.0.0/24 to any keep state group 100
pass in quick proto icmp from 192.168.0.0/24 to any keep state group 100
pass in quick proto esp from 192.168.0.0/24 to any keep state keep frags
group 100
pass in quick proto gre from 192.168.0.0/24 to any keep state group 100
```

As I said earlier on, we only have to deal with the first packet of each "connection". The rest is handled by the state table. The first rule permits TCP packets that only have the SYN flag set. This is the first packet that gets sent by a host on the LAN when it wants to establish a TCP connection. ipfilter will then remember the combination of source/destination IP addresses and ports. That way, we can enforce that each TCP connection has to begin with a SYN packet, so nasty things like FIN scanning are not possible anymore. Of course, WAN hosts won't be able to send any packets to us that are not part of a connection that was initiated by a LAN host at this time anyway, but you should not trust your LAN hosts too much, either.

The other 4 rules allow UDP, ICMP, ESP (for IPsec) and GRE (for PPTP).

Note that instead of 192.168.0.0/24 you could also specify 'any', but this provides additional security against spoofing (from internal hosts) and misconfigurations.

IPsec MTU worries...
ESP is required for IPsec connections, and we're using keep frags on that one because with most IPsec implementations you run into troubles if you try to run them over a PPPoE connection: they assume that they can send packets with a length of 1500 bytes (since they're on Ethernet), so they adjust the TCP MSS in a way that the final packet (with the ESP headers added) is exactly 1500 bytes. If you're now trying to squeeze that in the 1492 bytes that are possible with PPPoE connections, you run into troubles: you'll have to fragment the packet. ppp can silently adjust the MSS of normal TCP packets so the remote host will think that the LAN host actually requested an MTU of 1492 instead of 1500. This is not possible with ESP packets because they're encrypted.
We have to use "keep frags" so ipfilter will remember enough information to be able to let the other fragments pass through. This is not necessary for TCP and UDP since the MTU will be adjusted in a way that obliterates fragmentation (that's the way it should be).

Outgoing packets on the LAN interface
The only connections we're going to allow here are ICMP messages from the firewall host to hosts on the LAN interface. This makes it possible to ping LAN hosts from the firewall and enables the firewall to send other ICMP messages to LAN hosts (destination unreachable, etc.). Everything else is denied, so you cannot set up TCP/UDP connections from the firewall to LAN hosts. If you need this functionality, add the corresponding rules.

```
#------------------------------------------------------------------------
-
# outgoing traffic on LAN interface - group 150
#------------------------------------------------------------------------
-
pass out quick proto icmp from 192.168.0.254 to 192.168.0.0/24 keep state
group 150
```

Incoming packets on the WAN interface
If you don't have any DMZ or LAN hosts that provide services to machines on the internet (web/mail servers, for example), you don't need to allow any incoming packets on the WAN interface, as all legitimate packets (replies to outgoing connections) will be allowed by the rules in the state table. This means that you can just have the head rule for incoming WAN packets block eveything.

If you do need to make some ports available to external hosts (i.e. you're not going to block all incoming packets on the WAN interface), you should at least block packets from well-known non-

routable address spaces (like 192.168.x.x) because these are not really possible on the internet but can be used to fool your hosts into thinking that they actually came from a LAN host. Here's a possible ruleset

```
#------------------------------------------------------------------------
-
# incoming traffic on WAN interface - group 200
#------------------------------------------------------------------------
-
# block anything from private networks
block in quick from 10.0.0.0/8 to any group 200
block in quick from 127.0.0.0/8 to any group 200
block in quick from 172.16.0.0/12 to any group 200
block in quick from 192.168.0.0/16 to any group 200

# pass rules for mapped ports (to DMZ or LAN) go here
```

These are just the most common non-routable address spaces – if you're really paranoid, you may also wish to block the more exotic ones. Check out the ipfilter FAQ (see appendix A) for a more detailed list.

Outgoing packets on the WAN interface
If you don't need your firewall to be able to connect to hosts on the internet, you don't have to add any rules for this case, either. Unfortunately, ipfilter does not (yet) provide any way to check if a packet originated from the firewall machine, and since your WAN IP address is most probably dynamic, you cannot use it to verify this, either. At the moment, the only way is to allow all outgoing packets on the WAN interface:

```
#------------------------------------------------------------------------
-
# outgoing traffic on WAN interface - group 250
#------------------------------------------------------------------------
-
pass out quick proto tcp from any to any keep state group 250
pass out quick proto udp from any to any keep state group 250
pass out quick proto icmp from any to any keep state group 250
```

Finally, we'll add two default rules that block everything just to be sure that our default-to-deny policy is enforced. These rules may be hit by packets that go through interfaces that are not dealt with by our head rules:

```
#------------------------------------------------------------------------
-
# default rules (just to be sure)
#------------------------------------------------------------------------
-
block in quick all
block out quick all
```

That's it. Make sure you understand the meaning of all these rules before you load them!

```
ipf -Fa -f /etc/ipf.rules
```

Now it's time to check if your rules work the way you intended them to. Check if everything that should be allowed really works, but make sure everything that should be denied actually is, as well.

---

6. Adding rules for the DMZ interface (optional)

If you have hosts that provide services to the internet (like web/mail servers), it makes sense to put them in a separate network segment called a DMZ (Demilitarized Zone). The advantage over putting them in the LAN is that if they get compromised, the attacker won't be able to access other hosts on the LAN. The standard basic rules for a firewall with three interfaces, LAN, WAN and DMZ look like this:

allow LAN -> WAN
allow LAN -> DMZ
allow DMZ -> WAN (you may not even want/need this)
allow WAN -> DMZ on selected ports
deny everything else

Let's assume our DMZ interface is sis2 and the DMZ subnet is 192.168.1.0/24. We have to add a new group head rule first:

```
#------------------------------------------------------------------------
-
# group head 300 - DMZ interface
#------------------------------------------------------------------------
-
block in quick on sis2 all head 300
block out quick on sis2 all
```

And now for the group rules – incoming only as we don't need to pass any outgoing packets on the DMZ interface, except in case your firewall host needs to connect to DMZ machines. For connections from WAN -> DMZ that we allow, we'll add a keep state rule to the incoming WAN group.

```
#------------------------------------------------------------------------
-
# incoming traffic on DMZ interface - group 300
#------------------------------------------------------------------------
-
pass in quick proto tcp from 192.168.1.0/24 to ! 192.168.0.0/24 flags
S/SAFR keep state group 300
pass in quick proto udp from 192.168.1.0/24 to ! 192.168.0.0/24 keep state
group 300
pass in quick proto icmp from 192.168.1.0/24 to ! 192.168.0.0/24 keep state
group 300
```

We allow all hosts in the DMZ to set up connections, but not to hosts on the LAN. Say we have a server, 192.168.1.1, which provides SSH, SMTP and HTTP services to the outside. We have to add the following rules to the incoming WAN group:

```
pass in quick proto tcp from any to 192.168.1.1 port = 22 keep state group
200
pass in quick proto tcp from any to 192.168.1.1 port = 25 keep state group
200
pass in quick proto tcp from any to 192.168.1.1 port = 80 keep state group
200
```

Port ranges can be specified as well: say you wanted to map ports 7000-7010, you'd have to write "`port 6999 >< 7011`" instead of port = xx. That's a bit confusing, but "x >< y" in ipf rule language means "greater than x, but smaller than y", so it does not include the ports you specify.

We have to tell ipnat about this port mapping, too. Add the following lines to /etc/ipnat.rules:

```
rdr tun0 0/0 port 22 -> 192.168.1.1 port 22
rdr tun0 0/0 port 25 -> 192.168.1.1 port 25
rdr tun0 0/0 port 80 -> 192.168.1.1 port 80
```

Make sure the external interface name (tun0 in this example) is correct. Port ranges can be specified here as well, but in a more natural fashion than with ipf. Say we wanted to map ports 7000-7010, we'd just write:

```
rdr tun0 0/0 port 7000-7010 -> 192.168.1.1 port 7000
```

You don't have to specify the port range again in the end, just the first port (in fact, if you do, you'll get an error). It's even possible to map a range to another one – if you specified 9000 instead of 7000 in the end of that line, it would mean to ipnat that you wanted to translate all outside ports between 7000 and 7010 to the respective ports in the range 9000-9010 on 192.168.1.1.

---

7. Loading the rules

You can reload the ipnat and ipf rulesets now:

```
ipnat -C -f /etc/ipnat.rules
ipf -Fa -f /etc/ipf.rules
```

Be sure to add the following lines to /etc/rc.conf to ensure that the rules are automatically loaded on reboot:

```
ipnat_enable="YES"
ipfilter_enable="YES"
```

That's it.

---

8. Logging

Once you've got ipfilter up and running, it can be interesting to find out what is being blocked. Just add the keyword 'log' to rules which you find interesting (the block in all on WAN-interface is an interesting place to do this), reload the ruleset and run `ipmon`. Be careful not to log too much, however, as this may degrade performance. There's a logging buffer of 4 KB (IIRC), so you may already see some output. The manpage to ipmon(8) will tell you how to interpret the results.

If you want to see some statistics, try `ipfstat` or `ipfstat -s`. You can run `ipfstat -t` to see a top-style listing of entries in the state table.

---

9. Performance

There's one thing I have to admit, however. It seems like ipfilter is a tad slower than ipfw – as long as you don't count packets going through NAT, where the kernel-based ipnat gains speed over the userland natd. This is without keep-state rules in ipfw, however! If you count everything, their speed is probably about the same (although they claim that ipfw2 is much faster now). I have made some

measurements on my Soekris net4501 (only 486/133, remember), and the sustained throughput I can get from LAN <--> DMZ is about 20 Mbps (tested using iperf). This is more than I'll ever need on my firewall box (after all the internet connection is just 512 kbps and the machines on the DMZ interface communicate over a powerline bridge which can't handle more than about 3 Mbps). If you use a modern PC (with P3/P4/Duron/Athlon), I assume the firewall performance will be higher than what you can get out of 100 Mbps Ethernet anyway. Hint: if you want performance, use good cards like Intel or 3com. Avoid stuff like Realtek 8139...