Installing and Administering IPFilter/9000

Edition 1

HP Networking

Customer Order Number: B9901-90001

Manufacturing Part Number: B9901-90001
0201

United States

© Copyright 2000 Hewlett-Packard Company

Legal Notices

The information in this document is subject to change without notice.

Hewlett-Packard makes no warranty of any kind with regard to this manual, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Hewlett-Packard shall not be held liable for errors contained herein or direct, indirect, special, incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Warranty. A copy of the specific warranty terms applicable to your Hewlett- Packard product and replacement parts can be obtained from your local Sales and Service Office.

Restricted Rights Legend. Use, duplication or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 for DOD agencies, and subparagraphs (c) (1) and (c) (2) of the Commercial Computer Software Restricted Rights clause at FAR 52.227-19 for other agencies.

HEWLETT-PACKARD COMPANY 3000 Hanover Street Palo Alto, California 94304 U.S.A.

Use of this manual and flexible disk(s) or tape cartridge(s) supplied for this pack is restricted to this product only. Additional copies of the programs may be made for security and back-up purposes only. Resale of the programs in their present form or with alterations, is expressly prohibited.

The HP-UX Runtime Environment for Java is not designed or intended for use in on-line control of aircraft, air traffic, aircraft navigation, or aircraft communications; or in the design, construction, operation or maintenance of any nuclear facility.

Copyright Notices. ©copyright 1983-97 Hewlett-Packard Company, all rights reserved.

Reproduction, adaptation, or translation of this document without prior written permission is prohibited, except as allowed under the copyright laws.

©copyright 2000 IPFilter Based Firewalls HOWTO by Brendan Conoboy and Erik Fichtner.

Contents

1. Installing and Configuring IPFilter/9000

Overview of HP IPFilter/9000 Installation 11
Installation and Configuration Checklist 11

Step 1: Checking IPFilter/9000 Installation Prerequisites 13

Step 2: Loading HP IPFilter/9000 Software 14

Step 3: Determining the Rules for IPFilter 16

Step 4: Adding IPFilter Rules 17

Step 5: Loading Rules to the IPFilter Rules File 18

Step 6: Verifying the Installation and Configuration 20

Troubleshooting IPFilter/9000 21

2. Basic Firewalling

Configuration File Dynamics 25

Basic Rules Processing 26

Controlling Rule Processing 28

Basic Filtering by IP Address 29

Controlling Your Interfaces 31

Using IP Address and Interface Together 32

Bi-Directional Filtering; The out Keyword 34

Logging What Happens; The log Keyword 35

Complete Bi-Directional Filtering by Interface .36

Controlling Specific Protocols; The proto Keyword. 38

Filtering ICMP with the icmp-type Keyword; Merging Rulesets 39

TCP and UDP Ports; The port Keyword 40

Contents

3. Advanced Firewalling

The Default-Deny Stance 43

Implicit Allow; The keep state Rule 44
keeping state 44
Running an ssh server 45

Stateful UDP 47

Stateful ICMP. 48

FIN Scan Detection; flags Keyword, keep frags Keyword 49

Responding to a Blocked Packet 51

Logging Techniques 53

Putting It All Together. 54

Improving Performance With Rule Groups 55

Spoofing Services 58

Transparent Proxy Support; Redirection Made Useful 59

Keep State With Servers and Flags 60

4. IPFilter Utilities

Loading and Manipulating Filter Rules; The ipf Utility 63

The ipfstat Utility. 64

The ipmon Utility 67

The ipftest Utility. 69

5. IPFilter and FTP

Coping with FTP 75

Running an FTP Server 76

Contents

Active FTP 76
Passive FTP 76
Running an FTP Client 78

6. IPFilter Concepts

Localhost Filtering. 81
Drop-Safe Logging with dup-to and to 82
The dup-to Method 82
The to Method. 83

7. IPFilter and IPSec

IPFilter and IPSec Basics 87
IPSec UDP Negotiation. 89
Punching a Hole in Both Directions 91
When Traffic Appears to be Blocked. 92
Allowing Protocol 50 and Protocol 51 Traffic 94
IPSec Gateways 96

A. IPFilter Configuration Examples

BASIC_1.FW 99
BASIC_2.FW 102
example.1 104
example.2 105
example.3 106
example.4 107
example.5 108

Contents

example.6 109
example.7 110
example.8 111
example.9 112
example.10 113
example.11 114
example.12 115
example.13 116
example.sr. 117
firewall 119
ftp-proxy 120
ftpproxy. 122
server 123
tcpstate 124

Preface

The information in this manual is intended for network managers or network security administrators who install and administer IPFilter/9000.

This manual describes how to install, configure, and troubleshoot the HP IPFilter/9000 software product on HP 9000 systems.

NOTE

Almost all of the information in chapters 2 through 6 has been derived from the IP Filter-based Firewalls HOWTO document written by Brendan Conoby and Erik Fichtner. You can find this document at http://www.obfuscation.org/ipf/.

The manual is organized as follows:

Chapter 1	"Installing and Configuring IPFilter/9000" describes how to install and configure IPFilter/9000 software.
Chapter 2	"Basic Firewalling" describes basic IPFilter configuration. It also provides numerous example configurations.
Chapter 3	"Advanced Firewalling" provides information about IPFilter advanced configuration.
Chapter 4	"IPFilter Utilities" provides detailed information about IPFilter utilities. These utilities include the `ipf`, `ipfstat` and `ipmon` utilities.
Chapter 5	"IPFilter and FTP" provides information about running an FTP Server and an FTP Client.
Chapter 6	"IPFilter Concepts" provides a description of Drop-Safe logging.
Chapter 7	"IPFilter and IPSec" describes how IPFilter and IPSec work together.
Appendix A	"IPFilter Configuration Examples" provides twenty-one configuration examples.

1 Installing and Configuring IPFilter/9000

This chapter describes the procedures to load and configure HP IPFilter/9000 software on your system. It contains the following sections:

- Overview of HP IPFilter/9000 Installation.
- Step 1: Checking HP IPFilter/9000 Installation Prerequisites.
- Step 2: Loading HP IPFilter/9000 Software.
- Step 3: Determining the Rules for IPFilter/9000.
- Step 4: Adding IPFilter Rules.
- Step 5: Loading Rules to the IPFilter Rules File.
- Step 6: Verifying the Installation and Configuration.

Overview of HP IPFilter/9000 Installation

Installation of HP-UX IPFilter/9000 includes checking installation prerequisites and loading the HP-UX IPFilter/9000 filesets using the *swinstall(1M)* utility. The list in the next section summarizes each step in the process.

Installation and Configuration Checklist

The following checklist provides the sequence of steps that you will need to complete installation and configuration of IPFilter/9000. References to more in-depth information in this manual are also included as part of each step.

- Check that system meets the system prerequisites. Refer to "Checking IPFilter/9000 Installation Prerequisites" in this chapter for detailed information about this task.
- Install IPFilter/9000 using `swinstall`. Refer to "Loading HP IPFilter/9000 Software" in this chapter for detailed information about this task.
- Determine the rules for your system. Chapter 3 contains the rules for basic firewalling, Chapter 4 contains the rules for advanced firewalling and Appendix A contains examples of rulesets for specific problems. You should base your rules on the services running on your system.

NOTE The default rule for IPFilter is `pass all`.

- Add the filtering rules for your system to `/etc/opt/ipf/ipf.conf` file using the detailed information provided in "Step 4: Adding Your Rules to the IPFilter Configuration File."
- Load the rules into the IPFilter/9000 rules file.
- Run the `ipf` and `ipfstat` commands to verify the installation as described in "Step 6, Verifying the Installation and Configuration" in this chapter.

You can also refer to the `ipf (5)`, and `ipfstat(8)` man pages for more detailed information on these commands.

Step 1: Checking IPFilter/9000 Installation Prerequisites

1. Check that the operating system has been upgraded to HP-UX 11.0 or HP-UX 11.11. On HP-UX 11.0 systems, there is a dependency on the ARPA Transport 11.00 patch PHNE_22397 or greater. You may install IPFilter/9000 after the reboot following patch installation. Check the latest IPFilter/9000 release note for all other patch information.

   To obtain information about the OS, execute the command:

   ```
   uname -a
   ```

   To obtain information about a patch, execute the command:

   ```
   swlist -l patch <patch_id>
   ```

2. You have root access and are designated the network security administrator.

Step 2: Loading HP IPFilter/9000 Software

Follow the steps below to load HP-UX IPSec/9000 software using the HP-UX *swinstall* program.

NOTE If the product is downloaded to the system using

```
swinstall -s | <path to product depot>
```

follow Steps 5 - 12 below.

1. Log in as **`root.`**
2. Insert the software media (disk) into the appropriate drive.
3. Run the *swinstall* program using the command:

   ```
   swinstall
   ```

   This opens the Software Selection Window and Specify Source Window.
4. Change the Source Host Name if necessary, enter the mount point of the drive in the Source Depot Path field, and activate the **OK** button to return to the Software Selection Window. Activate the Help button to get more information.

   The Software Selection Window now contains a list of available software bundles to install.
5. Highlight the HP IPFilter/9000 software for your system type.
6. Choose `Mark for Install` from the “Actions” menu to choose the product to be installed. With an exception of the man pages and user’s manual, you must install the complete IPFilter product.
7. Choose `Install` from the “Actions” menu to begin product installation and open the Install Analysis Window.
8. Activate the **OK** button in the Install Analysis Window when the Status field displays a Ready message.

9. Activate the **Yes** button at the Confirmation Window to confirm that you want to install the software. *swinstall* displays the Install Window.

   View the Install Window to read processing data while the software is being installed. When the Status field indicates Ready and the Note Window opens.

   swinstall loads the fileset. Estimated time for processing: 3 to 5 minutes.

10. Activate the **OK** button on the Note Window to reboot the system.

    The user interface disappears and the system reboots.

11. After the system reboots, check the log files in */var/adm/sw/swinstall.log* and */var/adm/sw/swagent.log* to make sure the installation was successful.

NOTE Do not run the IPFilter/9000 product when the system is booted in single-user mode.

12. Go to “Step 3 Determining the Rules for IPFilter.”

Step 3: Determining the Rules for IPFilter

Review the IPFilter rule descriptions and examples in Chapter 3, Chapter 4 and Appendix A to determine the appropriate rules for your system. You should determine the rules that you use by the services that are running on your system.

Refer to Chapter 2, *Basic Firewalling* for examples and detailed information on the `block`, `pass`, `quick` and `log` keywords. Refer to Chapter 3, *Advance Firewalling* for information on the `keep state`, `flags`, `keep frags` and `rdr` keywords.

Go to “Step 4: Adding Rules to the IPFilter/9000 Configuration File.”

Step 4: Adding IPFilter Rules

When IPFilter is first installed, the default rules files `ipf.conf` and `ipfnat.conf` are empty. You must put rules into these files or change the configuration to read the files that hold these rules. After you have determined your IPFilter ruleset, add your rules to the `/etc/rc/config.d/ipfconf` file using a text editor such as `vi`.

Refer to the example rulesets in Appendix A for assistance in putting your ruleset together.

Go to “Step 5: Installing Rules in the IPFilter/9000 Configuration File.”

NOTE Although NAT functionality is included with the Hewlett-Packard product, Hewlett-Packard does not support NAT. For NAT support, contact the IPFilter public domain website.

Step 5: Loading Rules to the IPFilter Rules File

This section describes how to install rules in the IPFilter/9000 rules file.The IPFilter/9000 configuration file is named:

`/etc/opt/ipf/ipf.conf`

When IPFilter/9000 is installed, the `ipfconf` file is put in the `/etc/rc.config.d` directory. The information in this file determines how IPFilter/9000 will be started when the system is rebooted. The IPFilter/9000 `ipfboot` startup script reads `ipfconf`.

By default IPFilter/9000 will be started on bootup and the rules from the `/etc/opt/ipf/ipf.conf` file will be processed. NAT rules will be processed from the `/etc/opt/ipf/ipnat.conf`.

- Add new rules to your ruleset using the `-f` option of the `ipf` command:

  ```
  ipf -f <rules file>
  ```

NOTE Once a rule has been loaded, it takes effect immediately. For example, if you add a rule to block all traffic and load it using `ipf -f <rule file>`, you will find yourself blocked from X-windows and networking processes .

- Flush rules from your ruleset using the `-Fa` option of the `ipf` command,

  ```
  ipf -Fa
  ```

- The `-Fa` option will flush the previously configured rules.
  The `-d` option (debug mode) will check for possible errors.

  ```
  ipf -Fa -Ad -f /etc/opt/ipf/ipf.conf
  ```

- Optionally, use the `-I` option if you do not want to save previously configured rules. This command adds rules to the inactive rule list.

  ```
  ipf -I -Fa -Ad -f /etc/opt/ipf/ipf.conf
  ```

- This command enables the new rules. The option will swap the active rules you just configured with the inactive rules. To make the old rules effective again, use `ipf -s` to swap the rulesets.

  ```
  ipf -s
  ```

Go to"Step 6: Verifying the Installation and Configuration."

Step 6: Verifying the Installation and Configuration

Once your IPFilter/9000 software is installed, run the following commands to verify the installation and configuration.

- Verify that IPFilter/9000 is running using the `-V` option of the `ipf` command:

```
ipf -V
```

- Verify that IPFilter/9000 has been correctly loaded using the `kmadmin -s` command:

```
kmadmin -s

Name            ID      Status          Type
=====================================================
pfil            1       LOADED          STREAMS
ipf             2        LOADED           WSIO
```

- These commands will verify that your rules have been properly loaded. Run `ipfstat -i` to check for the inbound rules. Run `ipfstat -o` to check for outbound rules.

```
ipfstat -i
ipfstat -o
```

  To view all rules at the same time, run:

```
ipfstat -io
```

  IPFilter processes the rules in the `ipf.conf` file. The files that are provided with the product are empty and need to be loaded.

Troubleshooting IPFilter/9000

This section describes how to troubleshoot IPFilter/9000 configuration. It provides information about possible problems that may occur along with the steps needed to resolve them.

- **IPFilter/9000 is not working (It passes/allows all network traffic)**.

  Verify whether IPFilter/9000 is running using `ipf -V`. The running field should say `yes`. If it says `no`, then the IPFilter/9000 module has not been loaded. In fact, it was probably explicitly unloaded.

  To load IPFilter again:

  ```
  /sbin/init.d/ipfboot start
  ```

  This script will load the IPFilter module and any modules it depends on.

  `kmadmin -s` will provide information similar to the following.

  ```
  Name           ID     Status         Type
  ====================================================
  pfil           1      LOADED         STREAMS
  ipf            2       LOADED          WSIO
  ```

  Note the two in the information displayed above may vary depending on the number of DLKM modules configured on the system.

  Load the rules and check again that IPFilter works. If it still does not work, reboot the system and check `/etc/rc.log` and `/var/adm/syslog/syslog.log` for errors.

- **The host does not seem to be on the network and pings do not go through.**

  Check the rules you have configured using `ipfstat -io.` This command will show the `in` and the `out` rules. Note: If you are using `/etc/opt/ipf/ipf.conf` as your rules file, then it will be loaded at boot time. The IPFilter startup script `/sbin/init.d/ipfboot` will:

  — load the IPFilter module

  — start the logging daemon, `ipmon`

  — load any uncommented rules present in

```
/etc/opt/ipf/ipf.conf
```

If the last effective rule amounts to “block in all,” the boot sequence may not complete, for example, when `sendmail, snmp, NIS` are configured on the system.

- **Nothing is logged**.

  Verify the following:

  `ipf -V` should show the Logging file as `available`.

  `ps -ef|grep ipmon` to verify if `ipmon` is running. `ipmon` is started during bootup. If it is not running, start it as follows:

  ```
  ipmon -sD
  ```

  The `s` option specifies that the log records go to `/var/adm/syslog/syslog.log` and the `D` option directs `ipmon` to run as a daemon in the background.

- **Errors occur when loading rules**.

  ```
  # ipf -f <the rulefile>
  ioctl (add/insert rule); File Exists
  ```

  This occurs when you try to add a rule which is already loaded. Use the following command to load rules:

  ```
  ipf -Fa -f <the rulefile>
  ```

  The `-Fa` option will flush any previous rules present and all rules will be loaded afresh.

In addition, you can use `ipftest(1)` to test a set of filter rules without having to put them in place. Refer to the `ipftest(1)` for more information on this tool .

2 Basic Firewalling

This chapter describes the basic procedures to configure HP IPFilter/9000 software.

It contains the following sections:

- Configuration File Dynamics
- Basic Rule Processing.
- Controlling Rule Processing.
- Basic Filtering by IP Address
- Controlling Your Interfaces
- Using IP Address and Interface Together
- Bi-Directional Filtering; The `out` Keyword
- Logging What Happens; The `log` Keyword
- Complete Bi-Directional Filtering by Interface
- Controlling Specific Protocols; The `proto` Keyword
- Filtering ICMP with the `icmp-type` Keyword; Merging Rulesets
- TCP and UDP Ports; The `port` Keyword

NOTE Most of the information in this chapter has been derived from the IP Filter-based Firewalls HOWTO document written by Brendan Conoby and Erik Fichtner. You can find this document at http://www.obfuscation.org/ipf/.

Configuration File Dynamics

IPFilter has a configuration file that contains the IPFilter rules.

The UNIX configuration file conventions allow one rule per line, a "#" mark denoting a comment at the beginning of a line, and a rule and a comment on the same line. Extraneous whitespace is allowed and encouraged to keep the rules readable.

The IPFilter/9000 configuration file is named:

`/etc/opt/ipf/ipf.conf`

When IPFilter/9000 is installed, the `ipfconf` file is put in the `/etc/rc.config.d` directory. The information in this file determines how IPFilter/9000 will be started when the system is rebooted and also gvies the location of the rules file.

By default IPFilter/9000 will be started on bootup and the rules from the `/etc/opt/ipf/ipf.conf` file will be processed.

When IPFilter is first installed, the rules files are empty. You must put rules into these files or change the configuration to read the files that hold these rules. You can change the file information by editing the ipfilter rules file.

Refer to the example files in Appendix A for assistance in creating your IPFilter/9000 ruleset.

NOTE Although NAT functionality is included with the Hewlett-Packard product, Hewlett-Packard does not support NAT. For NAT support, contact the IPFilter public domain website.

Basic Rules Processing

Rules are processed in order from top to bottom. So, if the contents of your configuration file are,

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

the computer will process the rules as:

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

When a packet comes in, the first rule IPFilter applies is the first rule in the ruleset:

```
block in all
```

If IPFilter has a reason to move to the next rule, it would process the second rule:

```
pass  in all
```

When does IPFilter move on to the second rule? Many packet filters stop comparing packets to rulesets after the first match is made. IPFilter is not one of them.

Unlike other packet filters, IPFilter keeps a flag on whether or not it's going to pass the packet. Unless the flow is interrupted, IPFilter goes through the entire ruleset, making its decision on whether or not to pass or drop a packet based on the last matching rule.

Here is a possible scenario. IP Filter is running on your system:

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

A packet comes in the interface and the first rule is processed:

```
block in all
```

Based on the information in the first rule, the system blocks the packet. It then processes the second rule:

```
pass in all
```

The second rule indicates that the packet should be passed. It look for a third rule. As there is no third rule, the system goes with the specifications of the last rule and passes the packet.

It is important 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
```

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

Controlling Rule Processing

If you have had experience with other packet filters, you may find the IPFilter rules processing confusing. You may also have concerns about the portability with other filters and the speed of rule matching. Imagine if you had 100 rules and most of the applicable rules were the first ten. There would be considerable overhead for each packet in such a ruleset. Fortunately, there is a simple keyword you can add to any rule that causes it to take action if there is a match. That keyword is `quick`.

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

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

In this example, IPFilter looks at the first rule:

```
block in quick all
```

The packet matches and the search is over. The packet is not allowed to pass.

So, what happens with the next rule in the file?

```
pass  in       all
```

This rule is never encountered. The matching of `all` and the terminal keyword `quick` in the previous rule stopped the rule processing.

Having half a configuration file unused can negatively impact processing speed. On the other hand, the purpose of IPFilter is to block packets and, as configured, it's doing a good job. IPFilter, however, also lets some packets through, so a change to the ruleset to make this possible is also in order.

Basic Filtering by IP Address

IPFilter matches packets based on many different criteria. The most common one is the IP address. There are some blocks of address space from which traffic should never be received. 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. IPFilter recognizes both notations. If you want to block 192.168.0.0/16, here is one way to do it:

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

This will give you a less stringent ruleset that will do a bit more for you.

For example, imagine that a packet comes in with 1.2.3.4 as its source address. IPFilter applies the first rule:

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. IPFilter applies the second rule:

```
pass  in       all
```

The packet from 1.2.3.4 is definitely included in `all`, so the packet is sent on to its destination.

Now suppose you have a packet coming in from address 192.168.1.2. IPFilter applies the first rule:

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

There's a match, the packet is dropped, and that's the end of the processing. The second rule is not processed because the first rule is a match and it also contains the `quick` keyword.

At this point you can build an extensive set of definitive addresses that are passed or blocked. Since we're already blocking private address space from our firewall, lets complete the task:

```
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 in the example above are the unroutable IP space.

Controlling Your Interfaces

Your system may have interfaces to more than one network. You can control traffic based on the interface. Every packet you receive comes from a network interface; every packet you transmit goes out a network interface. If your machine has three interfaces, `lo0` (loopback), `lan0` (Ethernet), and `tun0` (FreeBSD's generic tunnel interface used by PPP), and you do not want packets coming in on the `tun0` interface, add the following rule to the configuration file.

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

In this case, the `on` keyword means 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 `lan0`, the first rule will not match, the second rule will, and the packet will be passed.

NOTE

Examples given in this chapter and following chapters use many different interface names. The interface names on your machine may be different from the interface names used in this manual. Check your configuration carefully.

Using IP Address and Interface Together

It is an odd state of affairs when one decides it is 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 becomes. If 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
```

Compare this to our previous ruleset:

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

In the previous ruleset, all traffic from 192.168.0.0/16, regardless of the interface, was completely blocked. With the new rule, `on tun0` means that a packet is only blocked if it comes in on the `tun0` interface. If a packet arrives on the `xl0` interface from 192.168.0.0/16, it is passed.

At this point you can build a set of definitive addresses that are passed or blocked. Since you've already started blocking private address space to keep it from entering `tun0`, lets take care of the rest of the traffic:

```
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
pass  in       all
```

You are already familiar with the first three blocks, but not with the fourth. The fourth is a wasted class-A network used for loopback. It is common for software to communicate with itself on 127.0.0.1, so blocking it from an external source is a good idea.

There is an important principle in packet filtering that has only been alluded to with the private network blocking and it is:

When you know there are certain types of data that only come from certain places, set up the system to only allow data from those places.

With 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 is an illegitimate packet. The same goes for the other unroutables as well as the 127.0.0.0/8 address.

Most software does all of its authentication based on 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 tun dialup, it's reasonable to drop it. It should not be allowed to get to its final destination. You can accomplish this with what you already know about IPFilter. The new ruleset will 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 an
block in quick on tun0 from 20.20.200/24 to any
pass in all
```

Bi-Directional Filtering; The *out* Keyword

In the previous examples, you've been passing or blocking inbound traffic. 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 packets coming in are not only filtered as they enter the firewall, they're also filtered as they exit. So far there's been an implied `pass out all`. Just as you may pass and block incoming traffic, you may do the same with outgoing traffic.

Now that you know there's a way to filter outbound packets, you can find an effective way to use this new rule. One possible use of this capability might be to limit permitted traffic to packets originating at 20.20.20.0/24.

To do so, add the following rules:

```
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 arrives from IP address 20.20.20.1/32, it is sent out by the first rule. If a packet comes from IP address 1.2.3.4/32, it is blocked by the second rule.

You can also make similar rules for unroutable addresses. If a machine routes a packet through IPFilter with a destination of 192.168.0.0/16, why not drop it? If so, 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
```

This won't enhance your security, but it does enhance the security of other systems. 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 probably find a number of reasons to block outbound packets. Do keep in mind is that `in` and `out` directions are in reference to your firewall system, never any other machine.

Logging What Happens; The `log` Keyword

In all of our examples so far, all blocked and passed packets have been silently blocked and silently passed. Sometimes, however, you may want to know if you're being attacked instead of always wondering if the firewall is really buying you any added benefits.

While you wouldn't want to log every passed packet and, in some cases, every blocked packet, you may want to know about the blocked packets from a specific address such as 20.20.20.0/24.

To do so, add the log keyword to the rule with that address:

```
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 log quick on tun0 from 20.20.20.0/24 to any
pass  in       all
```

Complete Bi-Directional Filtering by Interface

So far the example firewall has only included fragments of a complete ruleset. When you create a ruleset, you should setup rules for all directions and all interfaces. The default state of IPFilter is to pass packets. It is as though there is an invisible rule at the beginning of the ruleset that states `pass in all` and `pass out all`. Instead of relying on the IPFilter default behavior, make every ruleset as specific as possible, interface by interface, until all bases are covered.

In the continuation of the IPFilter example, the `lo0` loopback interface will be added. As these interfaces talk to other on the local system, leave these rules unrestricted:

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

The next rules are for the `xl0` interface. For now, no restrictions will be placed on the xl0 interface:

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

Finally, there’s the `tun0` interface, which was half-filtered in our previous firewall examples:

```
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
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 log quick on tun0 from 20.20.20.0/24 to any
pass  in     all
```

Future examples will continue to show the rules for one direction. When setting up your own ruleset, however, be sure that you add rules for all appropriate directions and interfaces.

Controlling Specific Protocols; The *proto* Keyword

Denial of Service attacks are rampant in many networks. Many denial of service attacks rely on glitches in the TCP/IP stack of the OS. Frequently, this has come in the form of ICMP packets. To block ICMP packets, add the `proto` command to your ruleset as follows:

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

In this example any ICMP traffic coming in from `tun0` will be logged and discarded.

Filtering ICMP with the *icmp-type* Keyword; Merging Rulesets

As dropping all ICMP packets may not be useful, you may want to keep some types of ICMP traffic and drop other types. If you want `ping` and `traceroute` to work, you will need to let in ICMP type 0 and type 11. Strictly speaking, this might not be a good idea, but if you need to weigh security against convenience, IPFilter will allow you to do that.

```
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
```

Rule order is important. As all rules are in `quick` mode, you should place your `pass` rules before your `block` rules. So, the last three rules should be 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 rules to the anti-spoofing rules created previously is tricky. One possibility would 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 log quick on tun0 from 20.20.20.0/24 to any
pass  in       all
```

The problem with this ordering 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.

In this new ordering of the rules, the ruleset will block spoofed traffic before the ICMP rules are processed. It's important to keep rule order in mind when merging rules.

TCP and UDP Ports; The *port* Keyword

In the last modification to the IPFilter example, you blocked packets based on protocol. Now you can block packets based on specific parts of a protocol. The most frequently used part is the port number. Services such as `rsh`, `rlogin`, and `telnet` are convenient to have, but are insecure against network sniffing and spoofing. One compromise is to block only the services externally and only allow them to run internally. This is easy to do because `rlogin`, `rsh`, and `telnet` use specific TCP ports (513, 514, and 23 respectively). Create rules to block these services as follows:

```
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 prototcp from any to 20.20.20.0/24 port = 23
```

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

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

To block UDP instead of TCP, replace `proto tcp` with `proto udp`. The rule for syslog would then be:

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

IPFilter also has a shorthand for rules that apply to `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

This chapter describes the advanced configuration procedures for IPFilter software. It contains concepts for advanced firewall design and

advanced features contained only within IPFilter.

It contains the following sections:

- The Default-Deny Stance
- Implicit Allow; The `keep state` Rule
- Stateful UDP
- Stateful ICMP
- FIN Scan Detection; `flags` Keyword, `keep frags` Keyword
- Responding to a Blocked Packet
- Fancy Logging Techniques
- Putting It All Together
- Improving Performance With Rule Groups
- Spoofing Services
- Transparent Proxy Support; Redirection Made Useful
- Keep State with Servers and Flags
- Asserted Kernel Variables

NOTE Most of the information in this chapter has been derived from the IP Filter-based Firewalls HOWTO document written by Brendan Conoby and Erik Fichtner. You can find this document at http://www.obfuscation.org/ipf/.

The Default-Deny Stance

A problem exists when blocking services by port as sometimes the port may move. This frequently happens with RPC-based programs such as `lockd`, `statd`, and even `nfsd` listen in places other than 2049. This is hard to predict and it is even harder to automate adjustments. What if you miss a service? Instead, let's start from the beginning with an empty ruleset.

The first rule in the new set is:

```
block in all
```

No network traffic will get through with this rule. While it is not an extremely useful ruleset, you're completely secure with this setup. It won't take much more to make your box secure and also useful.

In this example the machine is running on is a web server and it just wants to take connections on 80/tcp. You can set that up with a second rule as follows:

```
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 all other traffic. For basic firewalling, these are all the rules you need.

Implicit Allow; The *keep state* Rule

The purpose of a firewall is to prevent unwanted traffic from getting to point B from point A. There are general rules that say "as long as this packet is going to port 23, it's okay." There are also general rules that say "as long as this packet has its FIN flag set, it's okay." IPFilter firewalls don't know the beginning, middle, or end of TCP/UDP/ICMP sessions. There are only vague rules that are applied to all packets. You're left to hope that the packet with its FIN flag set isn't really a FIN scan mapping your services. You also hope that the packet to port 23 isn't an attempted hijack of your `telnet` session. If only there were a way to identify and authorize individual TCP/UDP/ICMP sessions and distinguish them from port scanners and DoS attacks. There is a way and it is called `keeping state`.

keeping state

In the expanded ruleset below, the goal is to have convenience and security in one ruleset. Some systems have an "established" clause that allows established `tcp` sessions to go through. `Ipfwadm` has setup established. They all have this feature. The name, however, is misleading. When you first saw the rule, you probably thought it meant the packet filter was keeping track of what was going on and that it recognized whether a connection had been established or not. The fact is, this feature takes this indicator from a part of the packet that can be misrepresented. IPFilter reads the flags for the packet's TCP section and that's the reason UDP/ICMP don't work. They have no such flag. Any user who creates packets with bogus flags can get by a firewall with this type of ruleset.

How will IPFilter rectify this situation. Unlike other firewalls, IPFilter is able to keep track of whether or not a connection has been established. And it will do it with TCP, UDP and ICMP, not just TCP. IPFilter refers to it as `keeping state`. The keyword used for this in the ruleset is `keep state`.

Until now this manual has described how packets come in and then how the ruleset checks them. When packets go out, the ruleset also checks them. Actually, what happens is: when packets come in, the state table gets checked and then *maybe* the inbound ruleset gets checked. When packets go out, the state table gets checked and then *maybe* the

outbound ruleset gets checked. The state table is a list of TCP/UDP/ICMP sessions that are passed through the firewall circumventing the entire ruleset. This may sound like a serious security hole. It may, however, be the best part of your firewall.

Running an ssh server

All TCP/IP sessions have a start, a middle, and an end even though they relate to the same packet. You cannot have an end without a middle and you cannot have a middle without a beginning. So, all you need to filter on is the beginning of the TCP/UDP/ICMP session. If the beginning of the session is allowed by your firewall rules, you want the middle and end to be allowed too. If not, your IP stack may overflow and your machines may become useless. `keeping state` will ignore the middle and the end of TCP/IP sessions and focus on blocking/passing new sessions. If a new TCP/IP session is passed, all subsequent packets will be allowed through. If it's blocked, none of the subsequent packets will be allowed through.

```
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
```

In the above example there is no `pass out` provision. In fact, there is 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 is 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 not running any services. In fact, it is 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 privilege 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 without the hinderance or inspection of the firewall ruleset.

These rules also work 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
```

So far this example demonstrates `keeping state` on TCP, UDP, ICMP. Now you can make outgoing connections as though there's no firewall and would be attackers will not be able to get back in. This is very useful because there's no need to track down which ports you're listening to. Instead you attach only the ports you want people to be able to get to.

While state is useful, it is also tricky. 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
```

This seems like a good setup. Incoming sessions are allowed to port 23 and outgoing sessions to any port. Naturally packets going to port 23 will have reply packets, but the ruleset is arranged so that the `pass out` rule generates a state entry and everything seems to work fine.

Unfortunately after 60 seconds of idle time the state entry closes (as opposed to the normal 5 days). This is because the state tracker did not see the original SYN packet destined to port 23. It only saw the SYN ACK. IPFilter is effective when following TCP sessions from start to finish, but it's not very successful when coming into the middle of a connection. Rewrite the rule as follows:

```
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
```

Adding this rule will cause the very first packet to be added to the state table. Other processing will work as expected. Once the 3-way handshake has been witnessed by the state engine, it is marked in 4/4 mode. This means it is setup for long-term data exchange until such time as the connection is torn down when the mode will change again. You can see the current modes of your state table using `ipfstat`

Stateful UDP

As UDP is stateless, it is naturally harder to `keep state` on it. Nevertheless, IPFilter is able to do so.

When machine A sends a UDP packet to machine B with source port X and destination port Y, IPFilter will allow a reply from machine B to machine A with source port Y and destination port X. This is a short term state 60 second entry.

This is an example of how you use `nslookup` to get the IP address of www.hp.com:

```
$ nslookup www.hp.com
```

The following DNS packet is generated by this command:

```
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 and is 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. The following packet is received a few 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 the same moment that packet is let through, the state gateway is closed and new incoming packets are not allowed through, even if they are from the same place of origin.

Stateful ICMP

There are two 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, such as 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. So, if you are keeping state on any outbound `icmp` message that might send an `icmp` message in reply, you need a `proto icmp [...] keep state` rule

The majority of ICMP messages are, however, status messages generated by a failure in UDP (and sometimes TCP) and in 3.4.x and greater IPFilter, 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 may have generated that message, the ICMP packet is let in. For example, in older IPFilters, if you wanted `traceroute` to work, you would have used:

```
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
```

Now you can keep state on `udp` with:

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

To provide protection against a third-party sneaking ICMP messages through your firewall when an active connection is known to be in your state table, check the incoming ICMP packet 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.

FIN Scan Detection; *flags* Keyword, *keep frags* Keyword

Lets go back to the four 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 satisfactory. The problem is that it's not only SYN packets that are allowed to go to port 23. Any packet can get through. You can change this 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 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). There are at least two advantages to this: no arbitrary packets can come in and negatively impact your state table, and FIN and XMAS scans will fail as they set flags other than the SYN flag. All incoming packets must either be handshakes or have state already. If any other packet comes in, it will probably be a port scan or a forged packet. There's one exception. That is when a packet comes in that's fragmented from its journey. The IPFilter provision for this is the `keep frags` keyword. With it, IPFilter will notice and keep track of packets that are fragmented, allowing the expected fragments to pass through. In this example the three rules are rewritten to log forgeries and allow fragments:

```
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 valid packet makes it into the state table before the blocking rules are reached. The only scan that is not detected in this scenario is SYN scan itself. If you're concerned about it, you may even want to log all initial SYN packets.

Some examples use flags S/SA instead of flags S. Flags S equates to flags S/AUPRFS and matches against only the SYN packet, out of all six

possible flags, while flags S/SA allows packets that may or may not have the URG, PSH, FIN, or RST flags set. Some protocols demand the URG or PSH flags. S/SAFR would be a better choice for these protocols. It may be less secure to use S/SA when it isn't required.

Responding to a Blocked Packet

In the previous examples, blocked packets have been dumped on the floor, logged or not, and no reply has been sent back to the originating host. Sometimes this isn't the best response because by doing so, the attacker knows that a packet filter is present. An improvement would be to misguide the attacker into believing that, while there's no packet filter running, there are also no services to break in to. This is where more refined blocking becomes useful.

When a service isn't running on a Unix system, it normally notifies the remote host with a return packet. In TCP, this is done with an `RST` (Reset) packet. When blocking a TCP packet, IPFilter returns an RST packet to the origin when the `return-rst` keyword is used.

Past examples include the following:

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

The example now looks like this:

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

This example has two `block` statements since `return-rst` only works with TCP and it still wants to block protocols such as UDP and ICMP. When that this is done, the remote side receives a "connection refused" message instead of a "connection timed out" message.

It is also possible to send an error message when a packet is sent to a UDP port on your system. In previous examples you might have observed:

```
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
```

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, but `port-unreachable` is probably the best. It's also the default ICMP type for `return-icmp`.

When using `return-icmp`, you'll notice that it returns the ICMP packet

with the IP address of the firewall, not the original destination of the packet. This problem 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
```

Logging Techniques

The presence of the `log` keyword in your ruleset ensures that the packet will be available to the `ipfilter` logging device, `/dev/ipl`. To actually see this information, you must be running the `ipmon` utility or some other utility that reads from `/dev/ipl`. Users usually couple the usage of `log` with `ipmon -s` to log the information to `syslog`. You can 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
```

NOTE At times the longer lines in the example rulesets may wrap around to the next line. These lines have an ellipsis at the end of the line.

In addition, you can tailor the information being logged. For example, you may not be interested that someone attempted to probe your `telnet` port 500 times. You are, however, interested that they probed you once. You can use the `log first` keyword to only log the first example of a certain type of packet. "Firstness" only applies to packets in a specific session. For the typical blocked packet, it will be difficult to find situations where this keyword does what you expect. However, if you use it 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 other parts of the packet in addition to the usual header information. `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 may make your logs very verbose. For certain applications, it is often useful to be able to go back and take a look at the packet or to send this data to another application for further examination.

Putting It All Together

Now the firewall is fairly secure, but it can still be more secure. Some of the original rules in the original ruleset that were removed might be useful at this time. The ruleset will now look as follows:

```
by    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 log quick on tun0 from 20.20.20.0/24 to any
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
```

Improving Performance With Rule Groups

To improve performance, you could change the interface names and network numbers as shown in the next example. Let's assume that there are 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
```

Here is the entire ruleset:

```
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 log quick on xl0 from 20.20.20.0/24 to any
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
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 example, it becomes apparent that the ruleset is starting to become unwieldy. To make matters worse, as more rules are added to the DMZ network, we are adding additional tests that must be parsed for every packet, affecting the performance of the xl0 <-> xl2 connections. If you set up a firewall with a ruleset like this one, and you have a lot of bandwidth and a moderate amount of cpu, all your workstation users on the xl2 network are going to be unhappy. To prevent this situation, you can speed things up by creating rule groups. Rule groups allow you to

write your ruleset in a tree structure, instead of as a linear list, so that if your packet is unrelated to the set of tests (say, all those `xl1` rules) those rules will never be processed. It's equivalent to having multiple firewalls on the same machine.

Here's an example:

```
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 example, you can see a 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 will go on with the processing of the 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`. Using this technique you can rewrite the above rules to maximize the performance of your 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  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
```

Now you can see the rule groups in action. For a host on the `xl2` network, you can completely bypass all the checks in group 10 when

you're not communicating with hosts on that network.

Depending on your situation, it may be appropriate to group your rules by protocol, by machine, by netblock, or whatever makes the processing flow smoothly.

Spoofing Services

Lets imagine that there is a web server running on 20.20.20.5 and since you've gotten increasingly suspicious of your network security, you run this server on port 80 since that requires a brief lifespan as the root user. But how do you run it on a less privileged port of 8000? How will anyone find your server? You 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 point to 20.20.20.5:8000.

To do so, use the `rdr` keyword:

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

You can also specify the protocol here, if you want to redirect a UDP service, instead of a TCP service (which is the default). For example, you can move your entire network into one place with this rule:

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

You cannot easily use `rdr` as a "reflector." For example,

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

will not work when .5 and .6 are on the same LAN segment. The `rdr` function is applied to packets entering the firewall on the specified interface. When a packet comes in that matches an `rdr` rule, its destination address is rewritten, the packet is pushed into IPFilter for filtering and, if it is not blocked by the filter rules, it is sent to the Unix routing code. Since this packet is still inbound on the same interface that it will use to leave the system to reach a host, the system doesn't function properly. Reflectors do not work in this situation. It also doesn't work to specify the address of the interface the packet just came in on. `rdr` destinations must exit out the firewall host on a different interface than they arrive on.

Transparent Proxy Support; Redirection Made Useful

While installing your firewall, you may have decided that it is helpful to use a proxy for many of your outgoing connections. Then you can further tighten your filter rules protecting your internal network. This can be accomplished using 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 application of the `rdr` keyword is often more useful when you want to require users to authenticate themselves with the proxy. In this case you want your engineers to be able to access the web, but you would rather not have your call center staff doing so.

Keep State With Servers and Flags

Keeping state is useful, but it's easy to make a mistake when setting 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 that connection. One mistake that is made when mixing state tracking with filtering on flags is as follows:

```
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
```

This ruleset appears to allow a connection to be created to the `telnet` server on 20.20.20.20 with the replies going back. If you use this rule, you'll see that it does work--for a few moments. Since the rules are filtering for the SYN flag, the state entry never fully gets completed. The default time to live for an incomplete state is sixty seconds.

This can be solved 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
```

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.

4 IPFilter Utilities

This chapter describes IPFilter utilities. It contains the following sections:

- Loading and Manipulating Filter Rules
- Monitoring and Debugging
- The *ipfstat* Utility
- The *ipmon* Utility
- The *ipftest* Utility

NOTE Most of the information in this chapter has been derived from the IP Filter-based Firewalls HOWTO document written by Brendan Conoby and Erik Fichtner. You can find this document at http://www.obfuscation.org/ipf/.

Loading and Manipulating Filter Rules; The `ipf` Utility

IPFilter has two sets of rules, an active set and an inactive set. By default, all operations are performed on the active set. You can manipulate the inactive set by adding the `-I` option to the `ipf` command line. You can then toggle the two sets using the `-s` command line option. This is useful when you want to test new rule sets without wiping out the old ruleset.

You can also remove rules from the list using the `-r` command line option. It is usually safer, however, to flush the ruleset that you're working on with the `-F` option and completely reload it when you make changes.

In summary, the easiest way to load a ruleset is by entering a command similar to the following:

```
ipf -Fa -f <rules files>
```

For more complicated manipulations of the ruleset, refer to the `ipf(5)` `ipf(8)`man pages.

The `ipfstat` Utility

The `ipfstat` utility displays a table of data about your firewall performance, including how many packets have been passed or blocked, whether the packets were logged or not, and how many state entries have been made. Here's an example of the information you might see displayed after running this 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
```

The `ipfstat` utility can also provide information about your current ruleset. Using the `-i` flag or the `-o` flag will show the currently loaded rules for `in` or `out`, respectively. Adding a `-h` option to this command will provide more useful information while also showing you the "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 From to any keep state keep frags
```

From this we might conclude that the ruleset is not working as intended as there are a lot of blocked packets outbound in spite of a very permissive `pass out` rule. `ipfstat` will not indicate whether your rules are right or wrong. It can only show you what is happening at the present time with a given ruleset. To debug your ruleset, you may also want to run the utility with the `-n` flag set. The output 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
```

`ipfstat` can provide a dump of the state table. This is done by running the `ipfstat` utility 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
```

In this example there is one state entry for a TCP connection. The output will vary slightly from version to version, but the basic information is the same. There is a fully established connection, represented by the 4/4 state. Other states are incomplete and will be documented later. The state entry has a time life of 240 hours, which is an unusually long time. It is also the default for an established TCP connection. The TTL counter is decremented every second that the state entry is not used and will result in the connection being purged if it is left idle.

The TTL counter is reset to 864000 whenever the state is used, ensuring the entry will not timeout while it is being actively used. 196 packets consisting of about 17kB worth of data have been passed 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 numbers in the second line are the TCP sequence numbers for this connection. These numbers will help you ensure that someone isn't 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 generated by the `keep state` code showing that this connection is an inbound connection.

The ipmon Utility

`ipfstat` collects snapshots of what's happening with the system. It is often helpful, however, to have a tool that looks at events as they occur. `ipmon` has this capability. `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 these three. You can run this tool in the foreground or as a daemon which logs to syslog or a file. If you want to watch the state table in action, `ipmon -o S` will display the following:

```
# 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
```

In this case there is a state entry for an external `DNS` request off our nameserver, two `xntp` pings to well-known time servers, and a short-lived outbound `ssh` connection.

`ipmon` can also show us what packets have been logged. For example, when using `state`, you'll often run into the following packets:

```
# 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 output mean? The first field is a timestamp. The second field is the interface that this event occurred on. The third field is the rule that caused the event to happen.

Run `ipfstat -in` if you want to know which rule caused the problem. You use this command to look at rule 2 in rule group 0.

The fourth field, the little `b`, indicates that this packet was blocked. You can ignore this unless you're logging passed packets as well, which would show up as a little `p` instead.

The fifth and sixth fields are self-explanatory. They indicate where the packet came from and where it was going. The seventh (PR) and eighth fields display the protocol and the ninth field displays the size of the

packet. The last part, the `-A` in this case, displays the flags that were on the packet. This one was an ACK packet.

Why was state mentioned earlier? Due to the nature of the Internet, packets may be regenerated. Sometimes you'll receive two copies of the same packet and your state rule, which keeps track of sequence numbers, will have already seen this packet. If this occurs 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 had a chance to make it to your firewall. This is normal.

Here is example packet that might get 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. It is indicated by the ICMP type 9/0.

The `ipftest` Utility

The `ipftest` program tool can help you understand the actions that will be taken when you have IPFilter installed on your system. Generally it takes a set of rules (the same ones used with `ipf`) and applies them to a description of packets that simulate real traffic. The description of the packets can take many forms, as described in the `ipftest(1)` man page. The native format is described here. The final action taken by IPFilter is written out for every packet processed.

The advantage to using a tool such as `ipftest` is that it runs entirely in user space as a non-root program. No special privleges are necessary. This will allow you to test out a set of rules without compromising the security of the machine where the rules will eventually be placed.

For example use this rule set:

```
>>>Rule file start>>>
default
block in all

# particular test
pass in from 10.1.84.195 to any
<<< Rule file end <<<<<
```

For testing purposes, the following packets will be used to test this rule set:

```
>>>Packet file start>>>

in on lan0 udp 10.1.84.195,16000 10.1.84.196,16000
in on lan1 udp 10.1.84.195,16000 10.1.85.196,16000
in on lan0 udp 10.1.84.195,16000 10.1.80.196,16000

in on lan0 udp 10.1.85.195,16000 10.1.84.196,16000
in on lan1 udp 10.1.85.195,16000 10.1.85.196,16000
in on lan0 udp 10.1.85.195,16000 10.1.80.196,16000

out on lan0 udp 10.1.84.196,16000 10.1.84.195,16000
out on lan1 udp 10.1.85.196,16000 10.1.84.195,16000
out on lan0 udp 10.1.80.196,16000 10.1.84.195,16000

out on lan0 udp 10.1.84.196,16000 10.1.85.195,16000
out on lan1 udp 10.1.85.196,16000 10.1.85.195,16000
out on lan0 udp 10.1.80.196,16000 10.1.85.195,16000
```

```
in on lan0 udp 10.1.81.195,16000 10.1.84.196,16000
in on lan1 udp 10.1.81.195,16000 10.1.85.196,16000

out on lan0 udp 10.1.84.196,16000 10.1.81.195,16000
out on lan1 udp 10.1.85.196,16000 10.1.81.195,16000

out on lan0 icmp 10.1.84.196 10.1.84.195
in on lan0 icmp 10.1.84.195 10.1.84.196

out on lan0 udp 10.1.80.196,16001 10.1.84.195,16000
out on lan0 udp 10.1.80.196,16001 10.1.85.195,16000

in on lan0 udp 10.1.84.195,16000 10.1.80.196,16001
in on lan0 udp 10.1.85.195,16000 10.1.80.196,16001

<<< Packet file end <<<
```

There are many packets. These packets are similar to a test machine setup that is used in the actual testing of IPFilter. These packets are processed with the `ipftest` program and produce the following output using the command:

```
ipftest -r <rule set file> -i <packet file>
```

The name of the rules file is called `test01` in this case.

```
>>>

opening rule file "test01"
input: in on lan0 udp 10.1.84.195,16000 10.1.84.196,16000
pass ip 28(20) 17 10.1.84.195,16000 > 10.1.84.196,16000
--------------
input: in on lan1 udp 10.1.84.195,16000 10.1.85.196,16000
pass ip 28(20) 17 10.1.84.195,16000 > 10.1.85.196,16000
--------------
input: in on lan0 udp 10.1.84.195,16000 10.1.80.196,16000
pass ip 28(20) 17 10.1.84.195,16000 > 10.1.80.196,16000
--------------
input: in on lan0 udp 10.1.85.195,16000 10.1.84.196,16000
block ip 28(20) 17 10.1.85.195,16000 > 10.1.84.196,16000
--------------
input: in on lan1 udp 10.1.85.195,16000 10.1.85.196,16000
block ip 28(20) 17 10.1.85.195,16000 > 10.1.85.196,16000
--------------
input: in on lan0 udp 10.1.85.195,16000 10.1.80.196,16000
block ip 28(20) 17 10.1.85.195,16000 > 10.1.80.196,16000
--------------
input: out on lan0 udp 10.1.84.196,16000 10.1.84.195,16000
nomatch ip 28(20) 17 10.1.84.196,16000 > 10.1.84.195,16000
--------------
```

```
input: out on lan1 udp 10.1.85.196,16000 10.1.84.195,16000
nomatch ip 28(20) 17 10.1.85.196,16000 > 10.1.84.195,16000
--------------
input: out on lan0 udp 10.1.80.196,16000 10.1.84.195,16000
nomatch ip 28(20) 17 10.1.80.196,16000 > 10.1.84.195,16000
--------------
input: out on lan0 udp 10.1.84.196,16000 10.1.85.195,16000
nomatch ip 28(20) 17 10.1.84.196,16000 > 10.1.85.195,16000
--------------
input: out on lan1 udp 10.1.85.196,16000 10.1.85.195,16000
nomatch ip 28(20) 17 10.1.85.196,16000 > 10.1.85.195,16000
--------------
input: out on lan0 udp 10.1.80.196,16000 10.1.85.195,16000
nomatch ip 28(20) 17 10.1.80.196,16000 > 10.1.85.195,16000
--------------
input: in on lan0 udp 10.1.81.195,16000 10.1.84.196,16000
block ip 28(20) 17 10.1.81.195,16000 > 10.1.84.196,16000
--------------
input: in on lan1 udp 10.1.81.195,16000 10.1.85.196,16000
block ip 28(20) 17 10.1.81.195,16000 > 10.1.85.196,16000
--------------
input: out on lan0 udp 10.1.84.196,16000 10.1.81.195,16000
nomatch ip 28(20) 17 10.1.84.196,16000 > 10.1.81.195,16000
--------------
input: out on lan1 udp 10.1.85.196,16000 10.1.81.195,16000
nomatch ip 28(20) 17 10.1.85.196,16000 > 10.1.81.195,16000
--------------
input: out on lan0 icmp 10.1.84.196 10.1.84.195
nomatch ip 48(20) 1 10.1.84.196 > 10.1.84.195
--------------
input: in on lan0 icmp 10.1.84.195 10.1.84.196
pass ip 48(20) 1 10.1.84.195 > 10.1.84.196
--------------
input: out on lan0 udp 10.1.80.196,16001 10.1.84.195,16000
nomatch ip 28(20) 17 10.1.80.196,16001 > 10.1.84.195,16000
--------------
input: out on lan0 udp 10.1.80.196,16001 10.1.85.195,16000
nomatch ip 28(20) 17 10.1.80.196,16001 > 10.1.85.195,16000
--------------
input: in on lan0 udp 10.1.84.195,16000 10.1.80.196,16001
pass ip 28(20) 17 10.1.84.195,16000 > 10.1.80.196,16001
--------------
input: in on lan0 udp 10.1.85.195,16000 10.1.80.196,16001
block ip 28(20) 17 10.1.85.195,16000 > 10.1.80.196,16001

--------------
<<<
```

The resulting output indicates the processing that the kernel level filter would do on your "real system" had the rules been used. The results are one of three: `pass`, `block` or `nomatch` (in the HP released version of IPFilter, the default is `pass`. From the results you can verify that the filter SHOULD operate as expected. Obviously, this is a simple example. More complex examples can be created to reflect traffic that is actually encountered in a production environment. In addition, the rules would most likely be more complex to reflect the various connections used.

When generating simulated traffic, one can use example data obtained from a packet probe or other such monitors. These packets may show the specifics of the eventual traffic that will be encountered in the subject machine. You should be careful to include the various flags in TCP packets, as they are used in the various `keep state` rules.

For further information, consult the manual pages `ipftest(1)`, `ipf(8)`, and `ipf(5)`.

5 IPFilter and FTP

This chapter describes IPFilter and FTP. It contains the following sections:

- Coping with FTP
- Running an FTP Server
- Running an FTP Client

NOTE Most of the information in this chapter has been derived from the IP Filter-based Firewalls HOWTO document written by Brendan Conoby and Erik Fichtner. You can find this document at http://www.obfuscation.org/ipf/.

Coping with FTP

FTP has unique behavior that the firewall administrator must address. `ftp` client problems are different from `ftp` server problems. This chapter describes FTP server problems and FTP client problems.

The FTP protocol has two forms of data transfer: active and passive. Active data transfers are those where the server connects to an open port on the client to send data. Passive transfers are those where the client connects to the server to receive data.

Running an FTP Server

This section describes Active FTP and Passive FTP. Active FTP sessions are easy to set up on an FTP server. Passive FTP sessions are a challenge.

Active FTP

You should handle Active FTP sessions like an incoming HTTP or SMTP connection. Open the `ftp` port and use the `keep state` keyword to 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 on your `ftp` server on 20.20.20.20.

Passive FTP

As web browsers default to this mode, passive `ftp` is becoming fairly common and should be supported. The problem with passive connections is that for every passive connection, the server starts listening on a new port (usually above 1023). This is like creating a new unknown service on the server. Assuming you have a good firewall with a default-deny policy, the new service will be blocked, and Running FTP sessions will be broken.

```
pass in quick proto tcp from any to 20.20.20.20/32 port = 21  flags S keep state
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 will work, but it is not complete. When opening ports above 1023, a few other problems may be created. While 1-1023 is the designated area for server services to run, sometimes other programs also use port numbers higher than 1023, such as `nfsd` and `X`.

Fortunately the FTP server determines which ports are to be assigned active sessions. Instead of opening all ports above 1023, you can allocate ports 15001-19999 as `ftp` ports and only open up that range of your firewall. In `wu-ftpd`, you do this with the passive ports option in `ftpaccess`. The man page on `ftpaccess` for details on `wu-ftpd` configuration. For IPFilter, all you need to do is set up 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.
```

Running an FTP Client

As with FTP servers, there are two types of `ftp` client transfers: passive and active.

The simplest client transfer from the firewall is the passive transfer. If your server is keeping state on all outbound `tcp` sessions, passive transfers will work automatically. If your server is not doing this already, consider the following:

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

The client active transfer is a more complicated, but also more effective. Active transfers cause the server to open up a second connection back to the client for data to flow through. This is a problem when there is a firewall in the middle stopping outside connections from coming back in.

For more details on the IPFilter's internal proxies, see Transparent Proxy Support.

6 IPFilter Concepts

This chapter describes IPFilter concepts. It contains the following sections:

- Localhost Filtering

- What Firewall? Transparent Filtering
- Using Transparent Filtering to Fix Network Design Mistakes
- Drop-Safe Logging With `dup-to` and `to`

NOTE Most of the information in this chapter has been derived from the IP Filter-based Firewalls HOWTO document written by Brendan Conoby and Erik Fichtner. You can find this document at http://www.obfuscation.org/ipf/.

Localhost Filtering

The `tcp-wrapper` package adds a layer of protection to network services all over the world. Tcp-wrappers, however, have flaws. This is because tcp-wrappers only protect TCP services, as the name suggests. Also, unless you run your service from `inet` or you have compiled it with `libwrap` and the appropriate hooks, your service isn't protected. Large holes in your host security may result. You can plug these up using `ipf` on the local host as shown in the example below.

```
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 fragsI.
```

There has not been any negative impact resulting from running `ipf` all the time. If you want to tighten security, you could switch to the NAT `ftp` proxy and add rules to prevent spoofing. With these rules, this box is far more restrictive about what it presents to the local network and beyond than that presented by the typical host. This is useful if you run a machine that allows a lot of users on it and you want to make sure one of them doesn't start up a service that isn't allowed. 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 services safe even on a malicious LAN.

Using local host filtering in addition to a somewhat less restrictive "main firewall" machine can solve many performance issues as well as political user nightmares such as "Why doesn't ICQ work?" and "Why can't I put a web server on my own workstation! It's MY WORKSTATION!!" This solution allows you to have security and convenience at the same time.

Drop-Safe Logging with `dup-to` and `to`

In the examples so far, IPfilter has been used to drop packets. Instead of dropping them, you might pass them on to another system that can do more with this information beyond the logging performed with `ipmon`. The firewall system, whether it be a bridge or a router, can have as many interfaces as there are available on the system. You can use this information to create a "drop-safe" for your packets. You could also use this feature to implement an intrusion detection network. To begin, you might hide the presence of your intrusion detection system from the real network so that it is not detected.

In addition, there are some operational characteristics that should be noted. If you are only dealing with blocked packets, you can use the `to` keyword or the `fastroute` keyword. These keywords will be described later. If you're going to pass the packets as you usually do, you should make a copy of the packet for our drop-safe log with the `dup-to` keyword.

The `dup-to` Method

If, for example, you want to send a copy of every packet going out the xl3 interface off to your drop-safe network on `ed0`, you should include this rule in your filter list:

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

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

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

Note: this method will alter the copied packet's destination address. This may negatively impact the usefulness of the log. For this reason, we recommend that you use only the known address method of logging so you can be certain that the address that you're logging to corresponds in some way to the system for which you're logging. Don't use "192.168.254.2" for logging to your web server and your mail server, since, at a later time, you'll have a hard time trying to figure out which system was the target of a specific set of packets.)

This technique can be used effectively if you treat an IP Address on your drop-safe network in much the same way that you treat a multicast group on the real internet. In this case, "192.168.254.2" could be the channel for your http traffic analysis system and "23.23.23.23" could be your channel for `telnet` sessions. You don't need to actually have this address set as an address or alias on any of your analysis systems. Normally, your IPFilter machine will need to ARP for the new destination address using `dup-to ed0:192.168.254.2` style, but we can avoid this by creating a static `arp` entry for this "channel" on our IPFilter system.

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

The `to` Method

The `dup-to` method, however, does have an immediate drawback. As 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 this task and be ready to receive and process the next packet coming in to the IPFilter system.

If it is not important to pass the packet to its normal destination and you want to block it anyway, you can 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 lan0 to ed0 proto tcp from any to any port < 1024
```

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 cause your system to panic.

7 IPFilter and IPSec

This chapter describes how IPFilter/9000 and IPSec/9000 work together. It contains the following sections:

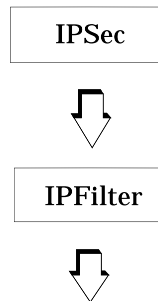
- IPFilter and IPSec Basics

- IPSec UDP Negotiation
- Punching a Hole in Both Directions
- When Traffic Appears to be Blocked
- Allowing Protocol 50 and Protocol 51 Traffic
- IPSec Gateways

IPFilter and IPSec Basics

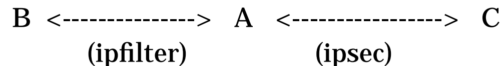
To use IPSec and IPFilter together, you must have an understanding of how the two products work together. While the products will not panic or corrupt each other, you do have to understand the situations in which one product might block traffic.

Figure 7-1



IPFilter, which is below IPSec in the networking stack, filters network packets. IPFilter also filters these packets before they reach IPSec. IPSec receives packets after IPFilter has decided whether it will pass or block them. You can have both IPFilter and IPSec configured and running on a machine without them negatively affecting each other.

Figure 7-2 **Scenario One**



For example, in Scenario One above you have IPFilter and IPSec on machine A with IPFilter blocking packets from machine B and IPSec encrypting packets from machine C. When a packet arrives at machine A, IPFilter checks to see if it is from machine B, and, if so, blocks the packet. If not, the packet continues up the stack to IPSec. IPSec checks to see if it is from machine C. If so, the packet arrives encrypted.

As there is no overlap in the configurations of IPFilter and IPSec in this network topology, there are no problems in Scenario One.

IPSec UDP Negotiation

If you have configured your IPSec and IPFilter products so that there is some overlap in the configurations, there are a few other details about IPSec that you should consider.

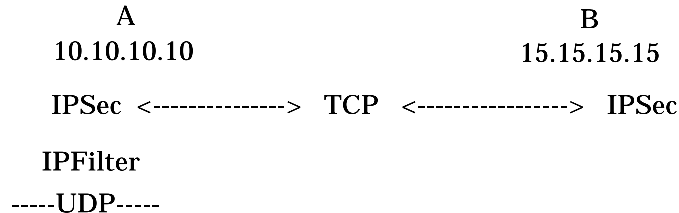
IPSec negotiates between two machines on a connection using Protocol UDP from Port 500 to Port 500.

And, if the IPFilter configuration is so broad that it is blocking the UDP traffic, then IPSec cannot complete it's negotiation. When the negotiation is not completed, the encrypted packets are not received. If this happens, you will see an IPSec error on the initiating side of "MM negotiation timeout."

To complete this negotiation, you may have to "punch a hole in your firewall" to let these packets through.

Lets go through an example.

Figure 7-3 Scenario Two



In Scenario Two, you want all UDP traffic to be blocked on machine A, you want all TCP traffic to pass through, and, from machine B on the network, you want all TCP traffic encrypted. Machine A has IP Address 10.10.10.10 and machine B has IP Address 15.15.15.15.

On machine A, you configure IPFilter to block all UDP traffic from all other machines as follows:

```
block in proto UDP
block out proto UDP
```

As the TCP traffic with machine B must by encrypted, you configure

IPSec on both machines using IPSec Manager. To do so, use the IP Addresses to specify that the TCP traffic is to be encrypted.

When TCP traffic is initiated from A to B or from B to A, the IPSec products on both machines will talk through a UDP/500 connection. You must punch a hole in IPFilter on machine A to let this traffic through. To do so, add the following commands to your configuration.

```
pass in quick proto UDP from 15.15.15.15 port = 500 to \ 10.10.10.10 port = 500
pass out quick proto UDP from 10.10.10.10 port = 500 to \ 15.15.15.15 port = 500
block in proto UDP
block out proto UDP
```

Scenario Two will now work correctly.

You could also setup this configuration by moving the UDP traffic block into IPSec. You would not use IPFilter in this situation.

Punching a Hole in Both Directions

Note: you MUST punch the hole in both directions.

If you let a UDP port 500 packet go out in IPFilter, a response can come back in during the next 60 seconds if the addressing is exactly reversed. This is done by punching the hole outward and entering "keep state":

```
pass out quick proto UDP from 10.10.10.10 port = 500 to \ 15.15.15.15 port = 500
keep state
```

If all IPSec connections are initiated outward, you may think that the UDP port 500 negotiation will always be initiated outward but this would be incorrect.

This causes a problem for IPSec. The negotiation of IPSec is a dialog that may resume at any point from either direction, even if the application you use always initiates in one direction.

So, when a hole is only punched outward, and a telnet session is initiated outward to machine B described in the last scenario, it worked fine at first.   Then, after 10 minutes when the Main Mode and Quick Mode SAs of IPSec had to be renegotiated, the dialog failed and the telnet session was unresponsive. It timed-out and IPSec logged the following error "MM negotiation timeout."

The hole MUST be punched in IPFilter so the dialog can be resumed in either direction at any time. For this to happen, you must punch the hole in both directions.

When Traffic Appears to be Blocked

In this scenario there is even more overlap in the configurations of IPFilter and IPSec.

To get this negotiation through, you must "punch a hole in your firewall" to let the TCP traffic through.

Figure 7-4 Scenario Three

```
        A                                      B
    10.10.10.10                           15.15.15.15

    IPSec <---------------> TCP <-----------------> IPSec

    IPFilter
 ---TCP-----
```

In Scenario Three, IPSec is configured to encrypt TCP traffic between machine A and machine B and IPFilter is configured to block all TCP traffic:

```
block in proto TCP
block out proto TCP
```

What will happen?

In this case an FTP session between A and B works fine.

When IPSec encrypts packets, it creates a new packet with a protocol number of 50. When it authenticates packets, it creates a new packet with a protocol number of 51.

Figure 7-5 Packet with Vanilla TCP Data

IP header (Protocol # = 6	TCP header	Data

Figure 7-6 Packet Where IPSec Has Encrypted TCP Data

IPFilter never sees the TCP packets between machine A and and machine B with a protocol number of 6. These packets are encrypted (or wrapped) in a packet that has a protocol number of 50. IPFilter was configured to block packets with protocol number 6, so it lets protocol number 50 pass through. Then IPSec will take apart the packet and unencrypt the TCP data.

Allowing Protocol 50 and Protocol 51 Traffic

If the IPFilter configuration is so broad that it blocks protocol 50 or protocol 51 traffic, then IPSec traffic will not get through.

Figure 7-7 **Scenario Four**

```
        A                                   B
   10.10.10.10                         15.15.15.15

   IPSec <---------------> TCP <-----------------> IPSec

  IPFilter
-----block !TCP-----
```

In Scenario Four, IPSec is configured to encrypt TCP traffic between the two machines and IPFilter is configured to block all traffic except TCP traffic. An IPSec UDP/500 hole has been correctly punched on machine B:

```
# IPSec hole with machine B
pass in quick proto UDP from 15.15.15.15 port 500 to \ 10.10.10.10 port = 500
pass out quick proto UDP from 10.10.10.10 port 500 to \ 15.15.15.15 port = 500
# Allow TCP traffic to/from anywhere
pass in quick proto TCP
pass out quick proto TCP
# Block all other traffic to/from anywhere
block in
block out
```

You might think that an FTP session between machine A and machine B would work. After all, both products have been configured to allow TCP traffic through.

Unfortunately, however, it doesn't work and there are no error messages in either IPSec or IPFilter that explain why this happens.

It occurs because IPSec encrypts the TCP traffic between the two machines and encrypted traffic has a protocol number of 50. These packets are being blocked by IPFilter!

To allow this traffic through IPFilter, add the following filter rules to

your configuration file:

```
pass in quick proto 50 from 15.15.15.15 to 10.10.10.10
pass out quick proto 50 from 10.10.10.10 to 15.15.15.15
```

Now the FTP session will work.

If IPSec is configured to do authentication, rather then encryption, the protocol 51 traffic must be let through.

IPSec Gateways

With the IPSec protocol there is the ability to configure encryption to a gateway as well as the end host with which you are communicating. The encryption to the gateway is called an IPSec Tunnel.

This poses no special problems, except that you will need to configure IPFilter to allow IPSec traffic with the gateway instead of the end node. To be more specific, the IPSec UDP/500 hole and protocol 50/51 traffic are passed to/from the gateway IP Address instead.

A IPFilter Configuration Examples

This appendix provides IPFilter configuration examples. These examples are also included in the `/opt/ipf/examples` directory with

the IPFilter/9000 product. You can take useful rules that you find in these examples and copy them into your IPFilter/9000 configuration file: `/etc/opt/ipf/ipf.conf.`

These files are taken from the files provided with the IPFilter freeware product.

BASIC_1.FW

```
#!/sbin/ipf -f -
#
# SAMPLE: RESTRICTIVE FILTER RULES
#
# ppp0 - (external) PPP connection to ISP, address a.b.c.d/32
#
# ed0 - (internal) network interface, address w.x.y.z/32
#
# This file contains the basic rules needed to construct a
# firewall for the above connections.
#
#-------------------------------------------------------
# Block short packets which are packets fragmented too short to
# be real packets.
block in log quick all with short
#-------------------------------------------------------
# Group setup.
# ============
# By default, block and log all packets.  This may result in
# too much information to be logged (especially for ed0)
# and needs to be further refined.
#
block in log on ppp0 all head 100
block in log proto tcp all flags S/SA head 101 group 100
block out log on ppp0 all head 150
block in log on ed0 from w.x.y.z/24 to any head 200
block in log proto tcp all flags S/SA head 201 group 200
block in log proto udp all head 202 group 200
block out log on ed0 all head 250
#-------------------------------------------------------
# Localhost packets.
# ==================
# Packets going in/out of network interfaces that aren’t on the
# loopback interface should *NOT* exist.
block in log quick from 127.0.0.0/8 to any group 100
block in log quick from any to 127.0.0.0/8 group 100
block in log quick from 127.0.0.0/8 to any group 200
block in log quick from any to 127.0.0.0/8 group 200
# Make sure the loopback allows packets to
# traverse it.
pass in quick on lo0 all
```

```
pass out quick on lo0 all
#-------------------------------------------------------
# Invalid Internet packets.
# ========================
#
# Deny reserved addresses.
#
block in log quick from 10.0.0.0/8 to any group 100
block in log quick from 192.168.0.0/16 to any group 100
block in log quick from 172.16.0.0/12 to any group 100
#
# Prevent IP spoofing.
#
block in log quick from a.b.c.d/24 to any group 100
#
#-------------------------------------------------------
# Allow outgoing DNS requests (no named on firewall)
#
pass in quick proto udp from any to any port = 53 keep ...
     state group 202
#
# If you are running named on the firewall and all internal
# hosts talk to it,use the following:
#
#pass in quick proto udp from any to w.x.y.z/32 port = 53 keep
#state group 202
#pass out quick on ppp0 proto udp from a.b.c.d/32 to any port =
#53 keep state
#
# Allow outgoing FTP from any internal host to any external FTP
# server.
#
pass in quick proto tcp from any to any port = ftp keep ...
     state group 201
pass in quick proto tcp from any to any port = ftp-data ...
     keep state group 201
pass in quick proto tcp from any port = ftp-data to any
     port > 1023 keep state group 101
#
# Allow NTP from any internal host to any external NTP server.
#
pass in quick proto udp from any to any port = ntp keep
     state group 202
#
# Allow outgoing connections: SSH, TELNET, WWW
#
```

```
pass in quick proto tcp from any to any port = 22 keep ...
     state group 201
pass in quick proto tcp from any to any port = telnet ...
     keep state group 201
pass in quick proto tcp from any to any port = www keep ...
     state group 201
#
#-------------------------------------------------------
block in log proto tcp from any to a.b.c.d/32 flags S/SA ...
     head 110 group 100
#
# Allow the following incoming packets types to the external
# firewall interface: mail, WWW, DNS
pass in log quick proto tcp from any to any port = smtp ...
     keep state group 110
pass in log quick proto tcp from any to any port = www keep...
     state group 110
pass in log quick proto tcp from any to any port = 53 keep ...
     state group 110
pass in log quick proto udp from any to any port = 53 keep ...
     state group 100
#-------------------------------------------------------
# Log these:
# ==========
# * Return RST packets for invalid SYN packets to help the
#other end close
block return-rst in log proto tcp from any to any flags ...
   S/SA group 100
# * Return ICMP error packets for invalid UDP packets
block return-icmp(net-unr) in proto udp all group 100
```

BASIC_2.FW

```
# SAMPLE: PERMISSIVE FILTER RULES
#
# ppp0 - (external) PPP connection to ISP, address a.b.c.d/32
#
# ed0 - (internal) network interface, address w.x.y.z/32
#
# This file contains the basic rules needed to construct a
# firewall for the above situation.
#
#-------------------------------------------------------
# Short packets which are packets fragmented too short to be
# real packets.
block in log quick all with short
#-------------------------------------------------------
# Group setup.
# ============
# By default, block and log all packets.  This may result in
# too much information to be logged (especially for ed0) and
# the rules needs to be further refined.
#
block in log on ppp0 all head 100
block out log on ppp0 all head 150
block in log on ed0 from w.x.y.z/24 to any head 200
block out log on ed0 all head 250
#-------------------------------------------------------
# Invalid Internet packets.
# =========================
#
# Deny reserved addresses.
#
block in log quick from 10.0.0.0/8 to any group 100
block in log quick from 192.168.0.0/16 to any group 100
block in log quick from 172.16.0.0/12 to any group 100
#
# Prevent IP spoofing.
#
block in log quick from a.b.c.d/24 to any group 100
#
#-------------------------------------------------------
# Localhost packets.
# ==================
# packets going in/out of network interfaces that aren’t on the
```

```
# loopbackinterface should *NOT* exist
block in log quick from 127.0.0.0/8 to any group 100
block in log quick from any to 127.0.0.0/8 group 100
block in log quick from 127.0.0.0/8 to any group 200
block in log quick from any to 127.0.0.0/8 group 200
# And of course, make sure the loopback allows packets to
# traverse it.
pass in quick on lo0 all
pass out quick on lo0 all
#---------------------------------------------------------
# Allow any communication between the inside network and the
# outside only.
#
# Allow all outgoing connections (SSH, TELNET, FTP, WWW,
# gopher, etc)
#
pass in log quick proto tcp all flags S/SA keep state group 200
#
# Support all UDP ‘connections’ initiated from inside.
#
# Allow ping out
#
pass in log quick proto icmp all keep state group 200
#---------------------------------------------------------
# Log these:
# ==========
# * return RST packets for invalid SYN packets to help the
# other end close
block return-rst in log proto tcp from any to any flags S/SA
group 100
# * return ICMP error packets for invalid UDP packets
block -icmp(net-unr) in proto udp all group 100
```

example.1

```
#
# block all incoming TCP packets on le0 from host 10.1.1.1 to
# any destination.
#
block in on le0 proto tcp from 10.1.1.1/32 to any
```

example.2

```
#
# block all outgoing TCP packets on le0 from any host to port
# 23 of host 10.1.1.2
#
block out on le0 proto tcp from any to 10.1.1.3/32 port = 23
```

example.3

```
# block all inbound packets.
#
block in from any to any
#
# pass through packets to and from localhost.
#
pass in from 127.0.0.1/32 to 127.0.0.1/32
#
# allow a variety of individual hosts to send any type of IP
# packet to any other host.
#
pass in from 10.1.3.1/32 to any
pass in from 10.1.3.2/32 to any
pass in from 10.1.3.3/32 to any
pass in from 10.1.3.4/32 to any
pass in from 10.1.3.5/32 to any
pass in from 10.1.0.13/32 to any
pass in from 10.1.1.1/32 to any
pass in from 10.1.2.1/32 to any
#
#
# block all outbound packets.
#
block out from any to any
#
# allow any packets destined for localhost out.
#
pass out from any to 127.0.0.1/32
#
# allow any host to send any IP packet out to a limited number
# of hosts.
#
pass out from any to 10.1.3.1/32
pass out from any to 10.1.3.2/32
pass out from any to 10.1.3.3/32
pass out from any to 10.1.3.4/32
pass out from any to 10.1.3.5/32
pass out from any to 10.1.0.13/32
pass out from any to 10.1.1.1/32
pass out from any to 10.1.2.1/32
```

example.4

```
#
# block all ICMP packets.
#
block in proto icmp from any to any
#
```

example.5

```
#
# test ruleset
#
# allow packets coming from foo to bar through.
#
pass in from 10.1.1.2 to 10.2.1.1
#
# allow any TCP packets from the same subnet as foo is on
# through to host 10.1.1.2 if they are destined for port 6667.
#
pass in proto tcp from 10.2.2.2/24 to 10.1.1.2/32 port = 6667
#
# allow in UDP packets that are NOT from port 53 and are
destined for localhost
#
pass in proto udp from 10.2.2.2 port != 53 to localhost
#
# block all ICMP unreachables.
#
block in proto icmp from any to any icmp-type unreach
#
# allow packets through that have a non-standard IP header
# length (ie there are IP options such as source-routing
# present).
#
pass in from any to any with ipopts
#
```

example.6

```
#
# block all TCP packets with only the SYN flag set (this is the
# first packet sent to establish a connection) out of the
# SYN-ACK pair.
#
block in proto tcp from any to any flags S/SA
```

example.7

```
# block all ICMP packets.
#
block in proto icmp all
#
# allow in ICMP echos and echo-replies.
#
pass in on le1 proto icmp from any to any icmp-type echo
pass in on le1 proto icmp from any to any icmp-type echorep
#
# block all ICMP destination unreachable packets which are
# port-unreachables
#
block in on le1 proto icmp from any to any icmp-type unreach
code 3
```

example.8

```
#
# block all incoming TCP connections but send back a TCP-RST
# for ones to the ident port
#
block in proto tcp from any to any flags S/SA
block return-rst in quick proto tcp from any to any port = 113
flags S/SA
#
# block all inbound UDP packets and send back an ICMP error.
#
block return-icmp in proto udp from any to any
```

example.9

```
# drop all packets without IP security options
#
block in all
pass in all with opt sec
#
# only allow packets in and out on le0 which are top secret
#
block out on le1 all
pass out on le1 all with opt sec-class topsecret
block in on le1 all
pass in on le1 all with opt sec-class topsecret

#
```

example.10

```
#
# pass ack packets (ie established connection)
#
pass in proto tcp from 10.1.0.0/16 port = 23 to 10.2.0.0/16 ...
     flags A/A
pass out proto tcp from 10.1.0.0/16 port = 23 to 10.2.0.0/16...
     flags A/A
#
# block incoming connection requests to my internal network
# from the internet.
#
block in on le0 proto tcp from any to 10.1.0.0/16 flags S/SA
#  block the replies:
block out on le0 proto tcp from 10.1.0.0 to any flags SA/SA
```

example.11

```
#
# allow any TCP packets from the same subnet as foo is on
# through to host 10.1.1.2 if they are destined for port 6667.
#
pass in proto tcp from 10.2.2.2/24 to 10.1.1.2/32 port = 6667
#
# allow in UDP packets which are NOT from port 53 and are
destined for localhost
#
pass in proto udp from 10.2.2.2 port != 53 to localhost
#
# block any packet trying to get to X terminal ports, X:0 to
# X:9
#
block in proto tcp from any to any port 5999 >< 6010
#
# allow any connections to be made,except to BSD
# print/r-services this will also protect syslog.
#
block in proto tcp/udp all
pass in proto tcp/udp from any to any port 512 <> 515
#
# allow any connections to be made, except to BSD
# print/r-services
# this will also protect syslog.
#
pass in proto tcp/udp all
block in proto tcp/udp from any to any port 511 >< 516
```

example.12

```
#
# get rid of all short IP fragments (too small for valid
comparison)
#
block in proto tcp all with short
#
# drop and log any IP packets with options set in them.
#
block in log all with ipopts
#
# log packets with BOTH ssrr and lsrr set
#
log in all with opt lsrr,ssrr
#
# drop any source routing options
#
block in quick all with opt lsrr
block in quick all with opt ssrr
```

example.13

```
#
# log all short TCP packets to qe3, with 10.3.3.3 as the
intended destination for the packet.
#
block in on qe0 to qe3:10.3.3.3 proto tcp all with short
#
# log all connection attempts for TCP
#
pass in on le0 dup-to le1:10.3.3.3 proto tcp all flags S/SA
#
# route all UDP packets through transparently.
#
pass in on ppp0 fastroute proto udp all
#
# route all ICMP packets to network 10 out through le1, to
10.3.3.1
#
pass in on le0 to le1:10.3.3.1 proto icmp all
```

example.sr

```
# log all inbound packets on le0 which has IP options present
# log in on le0 from any to any with ipopts
#
# block any inbound packets on le0 which are fragmented and
#"too short" to do any meaningful comparison on.  This actually
# only applies to TCP packets which can be missing the
# flags/ports (depending on which part of the fragment you
# see).
#
  block in log quick on le0 from any to any with short frag
#
# log all inbound TCP packets with the SYN flag (only) set
#  (NOTE: if it were an inbound TCP packet with the SYN flag
#set and it had IP options present, this rule and the above
#would cause it to be logged twice).
#
# log in on le0 proto tcp from any to any flags S/SA
#
# block and log any inbound ICMP unreachables
#
#  block in log on le0 proto icmp from any to any icmp-type
# unreach
#
# block and log any inbound UDP packets on le0 which are going
to port 2049 (the NFS port).
#
block in log on le0 proto udp from any to any port = 2049
#
# quickly allow any packets to/from a particular pair of hosts
#
pass in quick from any to 10.1.3.2/32
pass in quick from any to 10.1.0.13/32
pass in quick from 10.1.3.2/32 to any
pass in quick from 10.1.0.13/32 to any
#
# block (and stop matching) any packet with IP options present.
#
block in quick on le0 from any to any with ipopts
#
# allow any packet through
#
pass in from any to any
```

```
#
# block any inbound UDP packets destined for these subnets.
#
block in on le0 proto udp from any to 10.1.3.0/24
block in on le0 proto udp from any to 10.1.1.0/24
block in on le0 proto udp from any to 10.1.2.0/24
#
# block any inbound TCP packets with only the SYN flag set that
# are destined for these subnets.
#
block in on le0 proto tcp from any to 10.1.3.0/24 flags S/SA
block in on le0 proto tcp from any to 10.1.2.0/24 flags S/SA
block in on le0 proto tcp from any to 10.1.1.0/24 flags S/SA
#
# block any inbound ICMP packets destined for these subnets.
#
block in on le0 proto icmp from any to 10.1.3.0/24
block in on le0 proto icmp from any to 10.1.1.0/24
block in on le0 proto icmp from any to 10.1.2.0/24
```

firewall

```
#Configuring IP Filter for firewall usage.
==========================================

Step 1 - Block out "bad" IP packets.
------------------------------------

Run the perl script "mkfilters".  This will generate a list of
blocking rules which:
a) blocks all packets which might belong to an IP Spoofing
attack;
b) blocks all packets with IP options;
c) blocks all packets which have a length which is too short
for any legal packet;

Step 2 - Convert Network Security Policy to filter rules.
---------------------------------------------------------

Draw up a list of which services you want to allow users to use
on the Internet (e.g. WWW, ftp, etc).  Draw up a separate list
for what you want each host that is part of your firewall to be
allowed to do, including communication with internal hosts.

Step 3 - Create TCP "keep state" rules.
---------------------------------------

For each service that uses TCP, create a rule as follows:

pass in on <int-a> proto tcp from <int-net> to any port
<ext-service> flags S/SA keep state

where
* "int-a" is the internal interface of the firewall.  That is,
it is the closest to your internal network in terms of network
hops.

* "int-net" is the internal network IP# subnet address range.
This might be something like 10.1.0.0/16, or 128.33.1.0/24

* "ext-service" is the service to which you wish to connect or
if it doesn't have a proper name, a number can be used.  The
translation of "ext-service" as a name to a number is
controlled with the /etc/services file.
```

ftp-proxy

```
#How to setup FTP proxying using the built in proxy code.
========================================================

NOTE: Currently, the built-in FTP proxy is only available for
use with NAT (i.e. only if you're already using "map" rules
with ipnat).  It does support null-NAT mappings, that is, using
the proxy without changing the addresses.

Lets assume your network diagram looks something like this:


[host A]
   |a
---+-------------+----------
                 |b
             [host B]
                 |c
---+-------------+----------
   |d
[host C]

and IP Filter is running on host B.  If you want to proxy FTP
from A to C then you would do:

map int-c ipaddr-a/32 -> ip-addr-c-net/32 proxy port ftp
ftp/tcp

int-c = name of "interface c"
ipaddr-a = ip# of interface a
ipaddr-c-net = another ip# on the C-network (usually not the
same as the
interface).

e.g., if host A was 10.1.1.1, host B had two network interfaces
ed0 and vx0 which had IP#'s 10.1.1.2 and 203.45.67.89
respectively,and host C was  203.45.67.90, you would do:

map vx0 10.1.1.1/32 -> 203.45.67.91/32 proxy port ftp ftp/tcp

where:
ipaddr-a = 10.1.1.1
int-c = vx0
```

```
ipaddr-c-net = 203.45.67.91

The "map" rule for this proxy should precede any other NAT
rules you are using.
```

ftpproxy

```
#!/bin/sh
# The proxy part is as follows:
# proxy [port <portname>] <tag>/<protocol>
# the <tag> should match a tagname in the proxy table, as does
# the protocol.
# this format isn't finalized yet
echo "map ed0 0/0 -> 192.1.1.1/32 proxy port ftp ftp/tcp" |
/sbin/ipnat -f -
```

server

```
#
# For a network server, which has two interfaces, 128.1.40.1
#(le0) and 128.1.2.1 (le1), we want to block all IP spoofing .
# attacks.  le1 is connected to the majority of the network,
# while le0 is connected to a leaf subnet.
# We're not concerned about filtering individual services
#
#
pass in quick on le0 from 128.1.40.0/24 to any
block in log quick on le0 from any to any
block in log quick on le1 from 128.1.1.0/24 to any
pass in quick on le1 from any to any
```

tcpstate

```
#
# Only allow TCP packets in/out of le0 if there is an outgoing
# connection setup somewhere, waiting for it.
#
pass out quick on le0 proto tcp from any to any flags S/SAFR
keep state
block out on le0 proto tcp all
block in on le0 proto tcp all
#
# allow nameserver queries and replies to pass through, but no
# other UDP
#
pass out quick on le0 proto udp from any to any port = 53
keep state
block out on le0 proto udp all
block in on le0 proto udp all
```

Index

B

bidirectional filtering
  by interface, 36
  out keyword, 34
blocked packets, 51
blocked traffic
  correcting, 92

C

configuring
  file conventions, 18, 25
  quick keyword, 28
  rules processing, 26

D

dup-to method, 82

E

examples, 99

F

file
  configuration, 25
filtering
  bidirectional, 34
  by interface, 31
  by IP address, 29
  by IP address and interface, 32
  local host, 81
firewall
  basic configuration, 24
ftp, 75
ftp client
  active ftp, 78
  passive ftp, 78
ftp protocol, 75
ftp server
  active ftp, 76
  passive ftp, 76

I

icmp, 48
icmp and ssh server, 45
icmp-type keyword, 39
installing
  loading software, 14
  overview, 11
  prerequisites, 13
interface as filtering criteria, 31
ip address
  as filtering criteria, 29
ip address and interface
  as filter criteria, 32
ipf utility, 63
ipfilter
  and ipsec, 87
  configuration examples, 99
ipfstat utility, 64
ipmon utility, 67
IPSec
  gateway, 96
ipsec and ipfilter, 87
ipsec udp negotiation, 89

K

keep state keyword, 44
keeping state, 44
  icmp, 48
  udp, 47
  with servers and flags, 60
keywords
  flags, 49
  icmp-type, 39
  keep frags, 49
  keep state, 44
  log, 35, 53
  port, 40
  proto, 38
  quick, 28
  rdr, 58

L

loading software, 14
local host
  filtering, 81
log keyword, 35, 53
logging
  drop-safe, 82
  packets, 35

O

overview, installing, 11

P

port keyword, 40
proto keyword, 38
punching a hole, 91

Q

quick keyword, 28

R

rdr keyword, 58
redirection, 58
reporting problems, 47
rule groups, 55
rules processing, 26

S

software, loading, 14
ssh server, 45
swinstall(1M), 14
swlist(1M), 13

T

tcp and ssh server, 45
to method, 83
transparent proxy support, 59
tree structure, 55

Index

U

udp and ssh server, 45
uname(1), 13
utilities
  ipf, 63
  ipfstat, 64