

IPFW Primer

Abstract

[ipfw\(8\)](#) is a firewall application that comes standard with FreeBSD. This book provides an introduction to IPFW and its capabilities. These capabilities are demonstrated using QEMU virtual machines in real-world scenarios.

Content includes descriptions of the main features of [ipfw\(8\)](#) and lab examples on using IPFW as a firewall or a network gateway.

Like IPFW itself, this book is a work in progress.

Table of Contents

Acknowledgments	5
Preface	6
Shell Prompts	6
Typographic Conventions	6
Notes, Tips, Important Information, Warnings, and Examples	6
1. Introduction	8
1.1. Quick Start	9
2. IPFW Operation	10
2.1. Firewall Server Scripts	11
2.2. External VM Scripts	11
2.3. Loading IPFW	11
2.4. Initial Firewall Setup	12
3. IPFW Rules	16
3.1. Practical Ruleset Development	17
3.2. Dynamic Rules	20
3.2.1. Notes on Rule Numbering	21
3.3. Keywords	26
3.3.1. Protocols	26
3.3.2. Addresses	28
3.3.3. Ports	30
3.3.4. Prob	32
3.3.5. Sets	33
3.3.6. Tags	39
3.3.7. Logging	40
3.3.7.1. Method 1 – using ipfw0 , the IPFW pseudointerface	40
3.3.7.2. Method 2 – use syslogd	40
3.3.7.3. Using Method 1	41
3.3.7.4. Using Method 2	42
3.3.8. Reset	46
3.3.9. Tee	47
3.3.10. Unreach	47
3.3.11. Setdscp	49
3.3.12. Skipto	50
3.3.13. Divert	53
3.3.14. Other Protocols	58
3.3.15. Limit	58
3.3.16. Call and Return	60
3.3.17. Using uid and gid in rules	63

3.4. Lookup Tables	65
3.4.1. Creating Lookup Tables	66
3.4.2. Using Tables in Rules	68
3.4.2.1. Understanding the Word tablearg	68
3.4.2.2. More on flow tables	72
3.5. Stream Control Transport Protocol (SCTP)	75
3.5.1. SCTP Versions	75
3.5.2. SCTP Protocol Operation	76
3.5.3. Using the TSCTP Testing Tool on FreeBSD	77
3.5.4. Downloading and Building usrstcp Programs	80
3.5.5. Encapsulated Echo Server and Client with IPFW	81
4. IPFW Dummynet and Traffic Shaping	84
4.1. Measuring Default Throughput	85
4.2. IPFW Commands for Dummynet	87
4.2.1. Simple Pipe Configuration	87
4.2.2. Simple Pipe and Queue Configuration	93
4.2.3. Relationships	101
4.2.4. Dynamic Pipes	101
4.2.5. Other Pipe and Queue Commands	105
4.3. Adding Additional Virtual Machines	106
5. ipfw NAT	108
5.1. General Procedures for Working NAT Examples	108
5.2. Setting Up for Simple NAT	108
5.3. Setting Up for LSNAT	116
5.3.1. Setting up LSNAT- One address (10.10.10.10)	119
5.3.2. Engaging Multiple Hosts With LSNAT	121
6. IPv6 Network Address Translation (IPv6NAT)	125
6.1. Stateful NAT64 (NAT64LSN) With DNS64	126
6.1.1. Setting Up for NAT64 / DNS64	127
6.1.2. Setting Up the dnshost VM	129
6.1.3. Setting Up for Stateless NAT64 - NAT64STL	133
6.2. 464XLAT	135
7. Other Keywords	144
7.1. abort / abort6	144
7.2. mark / setmark	144
7.3. NPTv6	146
7.3.1. NPTv6 Setup	147
7.3.2. NPTv6 Testing	148
7.4. ipttl	149
7.4.1. ipttl Setup	149
7.4.2. ipttl Testing	150

7.5. tcpdatalen	152
7.6. verrevpath / versrcreach / antispoof	153
7.7. jail	157
7.7.1. Host-based Jail Networking	158
7.7.2. Virtual Network (VNET) Jail Networking	161
Appendix A: Appendix A: QEMU Setup	166
A.1. QEMU and VM Installation Process	167
A.1.1. Disabling Syslog Messages to the Console in the Virtual Machines	170
A.1.2. Adding and Managing Serial Console Access to the VMs	171
A.2. Using mkbr.sh for Bridge and Tap Setup	174
Appendix B: Appendix B: Scripts and Code for QEMU Lab	177
Appendix C: Appendix C: Networking References	223
Appendix D: Appendix D. Managing Serial Terminals with tmux and screen	225
D.1. Using tmux(1) for Managing Serial Terminals	225
D.2. Using screen(1) for Managing Serial Terminals	226
Appendix E: Appendix E: DNS Server Configuration	228
Index	238

Acknowledgments

Thanks to the FreeBSD developers who added and maintain IPFW.

Also, thanks to those who took the time to read early drafts of this document and offered many valuable comments and criticisms.

Attributions

Artwork from the following sources was used in preparing this book:

- Stylized **nginx** logo

Original image: [NGINX Logo](#)

Obtained from Wikimedia https://en.wikipedia.org/wiki/Nginx#/media/File:Nginx_logo.svg. Image is listed as in the Public Domain.

The image was converted to .PNG and also to .BMP formats for use in this book and accompanying code. Conversions were accomplished via ImageMagick's **convert** program.

- Stylized **bind 9** logo

Original image: [BIND 9's New Logo](#)

Obtained from <https://www.isc.org/blogs/bind-9s-new-logo/>

Logo used with permission from Internet Systems Consortium (ISC).

The image was converted to .PNG and also to .BMP formats for use in this book and accompanying code. Conversions were accomplished via ImageMagick's **convert** program.

- Stylized **IPv6** logo

Original image: [World IPv6 Launch Logo](#)

Obtained from Wikipedia: https://commons.wikimedia.org/wiki/File:World_IPv6_launch_logo.svg
Original CC 3.0 license: <https://creativecommons.org/licenses/by/3.0/deed.en> Updated CC 4.0 license: <https://creativecommons.org/licenses/by/4.0/deed.en>

The image was converted to .PNG and also to .BMP formats for use in this book and accompanying code. Conversions were accomplished via ImageMagick's **convert** program.

Use of the above artwork does not imply endorsement of any opinions expressed in this book.

All other artwork is original artwork by the author.

Preface

This page describes the conventions used in this book.

Shell Prompts

This table shows the default system prompt and superuser prompt. The examples use these prompts to indicate which type of user is running the example.

User	Prompt
Normal user	%
root	#

Typographic Conventions

This table shows various typographic conventions used throughout the text.

Meaning	Examples
Data, sysctls, things to note.	1234, net.inet.ip.forwarding, em0
The names of files.	Edit .login.
On-screen computer output. Output highlighting.	You have mail. Read it very closely.
What the user types, contrasted with on-screen computer output.	# ipfw add 100 check-state 00100 check-state :default
Manual page references.	Use su(1) to change user identity.
Emphasis levels	<i>Emphasis.</i> Stronger emphasis. <i>Strongest emphasis.</i>
Environment variables.	<code>\$HOME</code> is set to the user's home directory.

Notes, Tips, Important Information, Warnings, and Examples

Notes, warnings, and examples appear within the text.



Notes are represented like this, and contain information to take note of, as it may affect what the user does.



Tips are represented like this, and contain information helpful to the user, such as showing an easier way to do something.



Important information is represented like this. Typically, these show extra steps the user may need to take.



Warnings are represented like this, and contain information warning about possible damage if the instructions are not followed. This damage may be physical, to the hardware or the user, or it may be non-physical, such as the inadvertent deletion of important files.

Example 1. A Sample Example

Examples are represented like this, and typically contain examples showing a walkthrough, or the results of a particular action.

Chapter 1. Introduction

This book is about one of the native firewalls included with FreeBSD, [ipfw\(8\)](#) - the Internet Protocol FireWall. **ipfw** is designed to operate on a FreeBSD host with multiple network interfaces, to filter out unwanted traffic and pass through desired traffic. It does this based on a collection of rules (numbered, text based statements) that are entered into the system from the command line. This usage model is different from many other firewall products that employ Graphical User Interfaces (GUIs), or separate control programs. All **ipfw** statements are entered into the user shell, typically by a user with root privileges or access to root privilege by means of programs that elevate normal user privileges such as [sudo\(8\)](#) or [doas\(1\)](#).

ipfw reads network traffic from the interfaces it knows about and processes them inside the FreeBSD kernel. **ipfw** itself is a kernel module that can be either compiled into the kernel or loaded at run time. It includes a number of other kernel modules (**ipfw_nat**, **ipfw_nptv6**, etc.) many of which are discussed in this book.

A bird's-eye view of **ipfw** operation notes that:

1. Rules are organized into a sorted list based on a rule number
2. Packets entering the kernel from a network interface or leaving the kernel via a network interface are checked against the ruleset
3. Rules are checked one by one and the first rule that matches the packet characteristics wins - that is, **ipfw** accepts the packet for processing - allowing transit through the firewall, denying transit, updating a counter, or moving the packet into userspace for specialized processing.

The book makes frequent reference to the [ipfw\(8\)](#) manual page and the reader is advised to become familiar with the manual page alongside this book. There is also a section on **ipfw** in the [FreeBSD Handbook Page on IPFW](#). The intent with this book is to provide examples and informative material beyond the manual page and handbook to increase understanding and usage of **ipfw**.

Throughout this book are many examples of using **ipfw** with virtual machines to simulate actual hardware. These examples were developed with [QEMU](#) version 9.2.0. It is, of course, entirely possible to perform all the examples in this book with real hardware. QEMU provides a way to perform the examples without spending any money for hardware. In either case, some setup is required.



Note that QEMU command syntax with some of the examples may have changed slightly by the time this book becomes available. Use the latest QEMU release where possible, and check the QEMU documentation if the examples in this book do not work correctly.

Also used are a number of scripts that allow easy [if_bridge\(4\)](#) and [tap\(4\)](#) setup, virtual machine setup, and data transfer from external VMs to or through a firewall VM. In the early examples, data transfer is accomplished with the **netcat** program, specifically the version distributed with the [nmap](#) package (www.nmap.org). This version, [ncat\(1\)](#), has the best coverage of features that are used throughout the book. A familiarity with the man page for [ncat\(1\)](#) is helpful, but not required.

All scripts used in this book are found in [Appendix B](#) and published under the BSD 3-clause license.

The scripts are also available on the GitHub [IPFW Primer](#) page.

When copy/pasting examples, be aware that some desktop copy/paste functions add an extra space (or multiple spaces) to the end of a line, messing up the Unix continuation character convention ' ... \ ' at the end of a line. Ensure that the paste function does not introduce extra spaces at the end of the line.



The examples in this book involve passing data between interfaces on the host system. A running firewall on the host such as **pf**, **ipfw**, or **ipfilter** (also known as **ipf**) may interfere with data transfer, so ensure that any host system firewall is disabled. In addition, take any necessary steps to ensure that this does not compromise the security of the host.

1.1. Quick Start

Instructions for setting up all virtual machines (VMs) are found in [Appendix A](#).

In general you will need the following:

- Intel®/AMD® machine with a 64-bit processor. Any machine manufactured to recent Microsoft Windows® specifications should work. Processor speed will determine how responsive the virtual machines appear, so the faster, the better.
- At minimum, 8GB RAM. Each virtual machine is configured to use 1GB and for the first half of the book, only four VMs are used at the same time. In later chapters, the **jail1** VM, will require more memory (8GB).
- At least 50GB of free disk space to install all virtual machines. Each virtual machine uses 4GB and the **jail1** VM uses 12GB.

For the first half of this book, only four virtual machines are necessary - the **firewall** VM, and the **internal**, **external1** and **external2** VMs.

Additional detail, along with setup instructions for all virtual machines, is provided in [Appendix A](#).

Chapter 2. IPFW Operation

So far, the initial lab setup should be that shown in the figure below except for IP addressing. All the examples in this book use "Special Use Addresses" for both IPv4 and IPv6. Address references used are described [here for IPv4 Addresses](#) (RFC 5737) and [here for IPv6 Addresses](#) (RFC 3849).

Configure both VMs to use the addresses shown in the figure below. Once these addresses are in place on the VMs, it is unlikely that the VMs will be able to access sites on the Internet. The **203.0.113.0/24** network is considered non-routable by Internet service providers and major telecom carriers on the Internet. However, this does not really matter, since all the communications for this book are local to the VMs running on the FreeBSD host. Bottom line - if the VMs need to access the Internet, leave them on DHCP, but for the examples in this book, use manual addressing as shown in the figure below.



An easy, efficient way to rapidly change IP addresses is to edit `/etc/rc.conf` and modify the address lines for the individual interfaces. After saving the file, run **service netif restart** to restart the interfaces with the new addresses. Check and reset the default route if needed.

FreeBSD Host

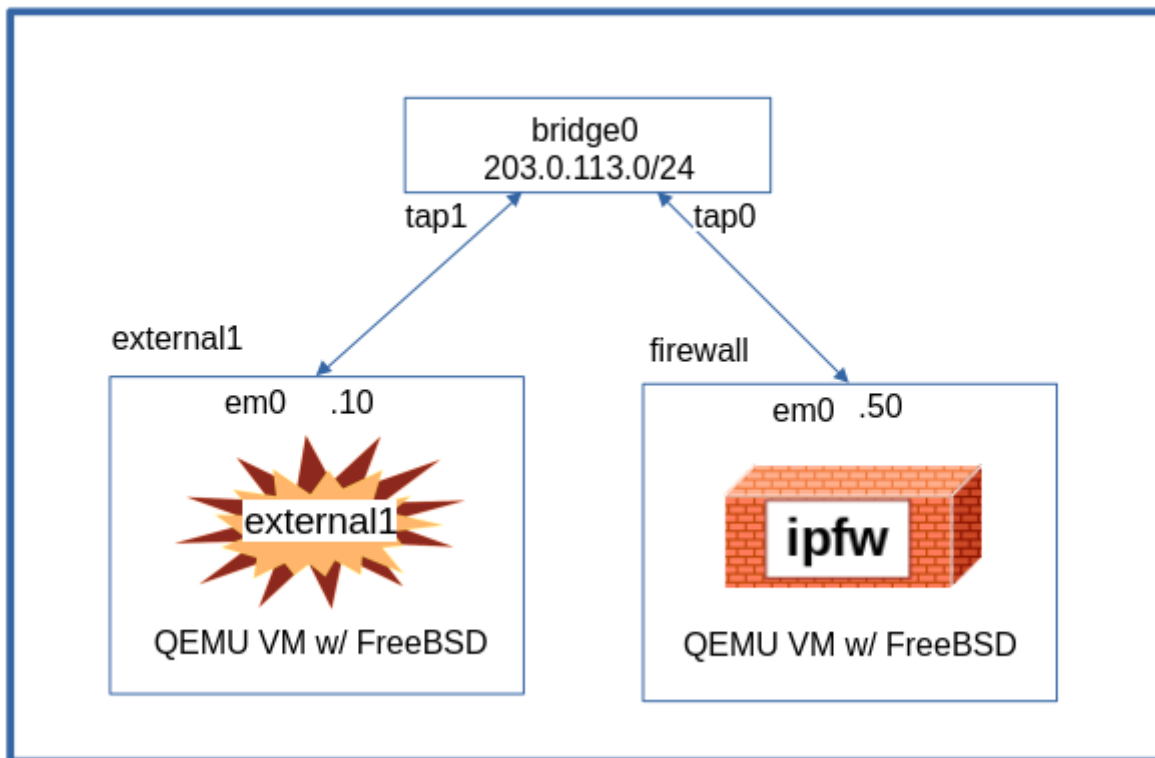


Figure 1. **external1**, and **firewall** VMs With Special Use Addresses

Setup instructions for the example:

```
% cd ~/ipfw-primer/ipfw/HOST_SCRIPTS
% sudo /bin/sh mkbr.sh reset bridge0 tap0 tap1
% /bin/sh runvm.sh firewall external1
```

2.1. Firewall Server Scripts

To demonstrate the firewall capabilities of **ipfw**, the **firewall** runs one of four basic scripts. Recall that these are located in `/root/bin`:

- **tserv.sh (userv.sh)**: Script that opens one TCP (UDP) port and listens for incoming traffic.
- **tserv3.sh (userv3.sh)**: Script that opens 3 TCP (UDP) ports and listens for incoming traffic.

These scripts will listen for an incoming connection and print whatever is sent over during the connection. When the connection is closed, the script will continue to listen for the next connection.

These script provide the basic mechanism for receiving a TCP or UDP connection. Once **ipfw** is running and populated with a ruleset, the effect each rule has on a connection can be shown.

2.2. External VM Scripts

Likewise, there are other basic scripts the **external1** (and later, **external2** and **external3**) VMs use for initiating or establishing communications with or through the **firewall** VM. The TCP and UDP versions perform similarly:

- **tcon.sh (ucon.sh)**: Connect via TCP (or UDP). This script takes a single argument, a port number to use for the connection. The external VM host can change the port number at each prompt. If there is no script listening on the port on the firewall, the script will indicate a "connection refused" or timeout error.
- **tconr.sh (uconr.sh)**: Connection takes a random port number and a sleep value. The script randomly selects one of three ports for its connection in a loop, controlled by the sleep value. If there is a listener on the firewall active on the port, the connection succeeds - otherwise the connection is refused.
- **tcont.sh (ucont.sh)**: Connection takes a port number and sleep value. The communication uses the same port in a loop, controlled by the sleep value.

These are simple scripts, but they allow for independent activity by the **external** VMs, while the **firewall** VM admin creates and tests **ipfw** rulesets. Most of the examples in the first part of this book can be done with just these scripts, so it is a good idea to become familiar with their operation. Later scripts will use [hping3\(8\)](#) and [iperf3\(1\)](#), versatile tools used in network analysis.

By default, the **external** VMs and **firewall** VM scripts work on ports **5656**, **5657**, and **5658**. The randomized communication scripts also utilize port **5659**, but in most cases, since no services are listening on that port on the **firewall** VM, the connection fails.

It is important to understanding the underlying network activity. If Internet protocols, network traffic, monitoring, and similar topics are unfamiliar, there are a number of excellent books, white papers, and tutorials, many free over the Internet. Check [Appendix C](#) for a modest selection.

2.3. Loading IPFW

ipfw can be built into the FreeBSD kernel directly, or it can be loaded as a kernel module. The

ipfw.ko loadable kernel module is used for most of the examples in this book. Load **ipfw** as root with the command:

```
# kldload ipfw
ipfw2 (+ipv6) initialized, divert loadable, nat loadable, default to deny, logging
disabled
```

Notice the kernel display output - "*ipfw2 (+ipv6) initialized, divert loadable, nat loadable, default to deny, logging disabled*". This gives a quick summary of this host's **ipfw** capabilities. The most important to note is "*default to deny*" which indicates that, by default, the firewall has an immutable rule located at the end of the ruleset that denies all Internet Protocol (IP) traffic. This rule depends on how the kernel was configured when it was built. By default it is "*default to deny*". However, when working on a FreeBSD system where its provenance is unknown, use the **ipfw list** command to make sure:

```
# ipfw list
65535 deny ip from any to any
#
```

Note that this does not mean it denies all network traffic, only traffic that is based on the Internet Protocol (RFC 791), and all of its derivatives (TCP, UDP, ICMP, etc). If a hacker had the capability to send and receive non-IP based traffic they could possibly still send and receive it. The firewall administrator would need special rules to deny all traffic, similar to the example later in this book.

To begin the next section, start with the **ipfw** firewall unloaded:

```
# kldunload ipfw
IP firewall unloaded
#
```



Unloading (**kldunload ipfw**) and loading (**kldload ipfw**) the **ipfw** kernel module is a handy way of completely re-initializing **ipfw**. This removes all **rules**, **sets**, **queues**, **pipes**, and other **ipfw** objects. See [kldload\(8\)](#), [kldunload\(8\)](#), and [kldstat\(8\)](#) for details.

2.4. Initial Firewall Setup

This section introduces the operation of the scripts described above and demonstrates simple traffic filtering.

In the first example, the firewall host runs **tserv.sh** (in `/root/bin`) which opens TCP port 5656.

External VM host	Firewall
<pre># sh tcon.sh sh tcon.sh PORTNUM # sh tcon.sh 5656 TCP connection from [203.0.113.50],[5656],[1] ncat [2] ready. Enter a valid PORTNUM: TCP connection from [203.0.113.50],[5656],[2] ncat [3] ready. Enter a valid PORTNUM: TCP connection from [203.0.113.50],[5656],[3] ncat [4] ready. Enter a valid PORTNUM: ^C #</pre>	<pre># sh tserv.sh Starting TCP listener on [5656] TCP connection from[203.0.113.10],[5656],[1] TCP connection from[203.0.113.10],[5656],[2] TCP connection from[203.0.113.10],[5656],[3] ^C#</pre>

Figure 2. Simple Transmit with No **ipfw** In Place

The **external1** VM runs **tcon.sh**, which repetitively opens a TCP connection and sends data to the **firewall** VM. Since there is no firewall in place, all TCP connections succeed.

On the host machine, it is possible to run **tcpdump(1)** on the **bridge0** device to see the traffic in real time. Shown below is one successful transfer, i.e there are no firewall rules preventing the connection.

It follows the basic TCP connection sequence: 3-way handshake setup, send data, and close the connection:


```

[root@host ~]# tcpdump -i bridge0 -X -vv
tcpdump: listening on bridge0, link-type EN10MB (Ethernet), capture size 262144 bytes
20:41:13.326513 IP (tos 0x0, ttl 64, id 0, offset 0, flags [DF], proto TCP (6), length 60)
 203.0.113.10.43331 > 203.0.113.50.5656: Flags [S], cksum 0x3976 (correct), seq 1934139331, win 65535,
options [mss 1460,nop,wscale 6,sackOK,TS val 3435594495 ecr 0], length 0
 0x0000: 4500 003c 0000 4000 4006 ce5f ac10 0a0a E..<..@.@.._....
 0x0010: ac10 0a32 a943 1618 7348 9fc3 0000 0000 ...2.C..sH.....
 0x0020: a002 ffff 3976 0000 0204 05b4 0103 0306 ....9v.....
 0x0030: 0402 080a ccc7 02ff 0000 0000 .....
20:41:13.327091 IP (tos 0x0, ttl 64, id 0, offset 0, flags [DF], proto TCP (6), length 60)
 203.0.113.50.5656 > 203.0.113.10.43331: Flags [S], cksum 0x939c (correct), seq 137438463, ack
1934139332, win 65535, options [mss 1460,nop,wscale 6,sackOK,TS val 307258952 ecr 3435594495], length 0
 0x0000: 4500 003c 0000 4000 4006 ce5f ac10 0a32 E..<..@.@.._....2
 0x0010: ac10 0a0a 1618 a943 0831 24ff 7348 9fc4 .....C.l$.sH..
 0x0020: a012 ffff 939c 0000 0204 05b4 0103 0306 .....
 0x0030: 0402 080a 1250 6648 ccc7 02ff .....PfH....
20:41:13.327766 IP (tos 0x0, ttl 64, id 0, offset 0, flags [DF], proto TCP (6), length 52)
 203.0.113.10.43331 > 203.0.113.50.5656: Flags [F], cksum 0xbe65 (correct), seq 1, ack 1, win 1026,
options [nop,nop,TS val 3435594495 ecr 307258952], length 0
 0x0000: 4500 0034 0000 4000 4006 ce67 ac10 0a0a E..4..@.@..g....
 0x0010: ac10 0a32 a943 1618 7348 9fc4 0831 2500 ...2.C..sH...1%.
 0x0020: 8010 0402 be65 0000 0101 080a ccc7 02ff .....e.....
 0x0030: 1250 6648 .....PfH
20:41:13.329214 IP (tos 0x0, ttl 64, id 0, offset 0, flags [DF], proto TCP (6), length 94)
 203.0.113.10.43331 > 203.0.113.50.5656: Flags [P], cksum 0xade7 (correct), seq 1:43, ack 1, win 1026,
options [nop,nop,TS val 3435594495 ecr 307258952], length 42
 0x0000: 4500 005e 0000 4000 4006 ce3d ac10 0a0a E..^..@.@..=....
 0x0010: ac10 0a32 a943 1618 7348 9fc4 0831 2500 ...2.C..sH...1%.
 0x0020: 8018 0402 ade7 0000 0101 080a ccc7 02ff .....
 0x0030: 1250 6648 5443 5020 6174 7461 636b 2066 .PfHTCP.communication.f
 0x0040: 726f 6d20 5b31 3732 2e31 362e 3130 2e31 rom.[203.0.113.1
 0x0050: 305d 2c5b 3536 5d2c 5b31 5d0a 0],[5656],[1].
20:41:13.329583 IP (tos 0x0, ttl 64, id 0, offset 0, flags [DF], proto TCP (6), length 52)
 203.0.113.10.43331 > 203.0.113.50.5656: Flags [F], cksum 0xbe3a (correct), seq 43, ack 1, win 1026,
options [nop,nop,TS val 3435594495 ecr 307258952], length 0
 0x0000: 4500 0034 0000 4000 4006 ce67 ac10 0a0a E..4..@.@..g....
 0x0010: ac10 0a32 a943 1618 7348 9fee 0831 2500 ...2.C..sH...1%.
 0x0020: 8011 0402 be3a 0000 0101 080a ccc7 02ff .....
 0x0030: 1250 6648 .....PfH
20:41:13.330266 IP (tos 0x0, ttl 64, id 0, offset 0, flags [DF], proto TCP (6), length 52)
 203.0.113.50.5656 > 203.0.113.10.43331: Flags [F], cksum 0xbe3b (correct), seq 1, ack 44, win 1025,
options [nop,nop,TS val 307258952 ecr 3435594495], length 0
 0x0000: 4500 0034 0000 4000 4006 ce67 ac10 0a32 E..4..@.@..g...2
 0x0010: ac10 0a0a 1618 a943 0831 2500 7348 9fef .....C.l%.sH..
 0x0020: 8010 0401 be3b 0000 0101 080a 1250 6648 .....;.....PfH
 0x0030: ccc7 02ff .....
20:41:13.331367 IP (tos 0x0, ttl 64, id 0, offset 0, flags [DF], proto TCP (6), length 52)
 203.0.113.50.5656 > 203.0.113.10.43331: Flags [F], cksum 0xbe3a (correct), seq 1, ack 44, win 1025,
options [nop,nop,TS val 307258952 ecr 3435594495], length 0
 0x0000: 4500 0034 0000 4000 4006 ce67 ac10 0a32 E..4..@.@..g...2
 0x0010: ac10 0a0a 1618 a943 0831 2500 7348 9fef .....C.l%.sH..
 0x0020: 8011 0401 be3a 0000 0101 080a 1250 6648 .....PfH
 0x0030: ccc7 02ff .....
20:41:13.333192 IP (tos 0x0, ttl 64, id 0, offset 0, flags [DF], proto TCP (6), length 52)
 203.0.113.10.43331 > 203.0.113.50.5656: Flags [F], cksum 0xbe2f (correct), seq 44, ack 2, win 1026,
options [nop,nop,TS val 3435594505 ecr 307258952], length 0
 0x0000: 4500 0034 0000 4000 4006 ce67 ac10 0a0a E..4..@.@..g....
 0x0010: ac10 0a32 a943 1618 7348 9fef 0831 2501 ...2.C..sH...1%.
 0x0020: 8010 0402 be2f 0000 0101 080a ccc7 0309 ...../.....
 0x0030: 1250 6648 .....PfH
^C

```

Figure 3. tcpdump(1) of Bridge Traffic During Transfer

Now load the **ipfw** firewall on the **firewall** VM and retry the communication.

```

# kldload ipfw
ipfw2 (+ipv6) initialized, divert loadable, nat loadable, default to deny, logging
disabled
#
# ipfw list
65535 deny ip from any to any
#

```

External VM host	Firewall
<pre># sh tcon.sh 5656 TCP connection from [203.0.113.50],[5656],[1] Ncat: TIMEOUT. ncat [2] ready. Enter a valid PORTNUM:(ret) TCP connection from [203.0.113.50],[5656],[2] Ncat: TIMEOUT. TCP Attack [203.0.113.10],[5656],[2] FAILED ncat [3] ready. Enter a valid PORTNUM:(ret) TCP connection from [203.0.113.50],[5656],[3] Ncat: TIMEOUT. TCP Attack [203.0.113.10],[5656],[3] FAILED ncat [4] ready. Enter a valid PORTNUM:(ret) ^C #</pre>	<pre># sh tserv.sh Starting TCP listener on [5656] ^C #</pre>

Figure 4. Simple Transmit with Default Rule In Place

No communications were successful - the connections time out because the **ipfw** firewall has denied traffic with the **default deny** rule as described above. The **external1** VM host sends SYN packets to start the connection, but they never reach the **firewall** VM's TCP service on port 5656. The TCP 3-way handshake is never completed.

These same techniques are used throughout this book. They show how communications and data transfer operate and how firewall rules affect those communications.

Chapter 3. IPFW Rules

The manual page for [ipfw\(8\)](#) lists the entire command syntax including those for general ruleset construction. In this book, basic traffic rules are discussed first and more complex capabilities are discussed in later sections. Each section builds on the previous sections so the reader will be better positioned to understand the advanced material later in the book.

Almost all the examples in this section can be run on the architecture used in the previous chapter, specifically that of [Chapter 2, Figure 1](#). See that section for guidance on setting things back up. Ensure IPv4 addressing is set up as shown in [Chapter 2, Figure 1](#).

ipfw rules have the general format of:

```
# ipfw command [rule_number] [set set_number] [prob match_probability] action [log
[logamount number]] [altq queue] [{tag | untag} number] body
```

Bolded keywords indicate literal option text that is added to a rule. Italicized keywords indicate a block of additional content that is rule dependent.

The rule body has its own syntax format:

```
[ proto from src to dst ] [ options ]
```

Here is an example with basic syntax for an entire rule:

```
# ipfw1 add2 10003 set 24 prob 0.55 deny6 log7 logamount 500008 altq red9 tag 2710 \  
tcp11 from12 any13 to14 me15 808116 //A contrived rule17
```

Figure 5. Description of **ipfw** Rule Syntax

1. The FreeBSD **ipfw** shell command
2. The **ipfw** rule command (add)
3. The optional rule number (1000)
4. An optional set keyword and value (set 2)
5. An optional probability keyword and value (prob 0.5)
6. The rule action keyword (deny)
7. An optional log keyword (log)
8. An optional logamount keyword and value (logamount 50000)
9. An optional altq keyword and value (altq red)
10. An optional tag keyword and value (tag 27)
11. The body of the rule starting with a protocol keyword (tcp)
12. A source direction keyword (from)

13. A source address (any internet address)
14. A destination keyword (to)
15. A destination address (any interface on the local system)
16. A destination port number (8081)
17. A comment. Note that comments are only valid in the match pattern section of a rule.

The example in the above figure would not actually load - there is no `altq(9)` queue named `red` set up yet - but it does show the overall format of how rules are constructed. This example also shows the use of the Unix line continuation convention using a single backslash at the end of the line (with no following spaces) to continue to the next line.

The essence of **ipfw** rule processing is found when patterns that are defined in the rule body are matched on incoming or outgoing packets, one at a time and the action keywords are processed in turn.

In other words, **ipfw** directs traffic flow by first matching each incoming or outgoing packet against patterns supplied within the body of the rule. The patterns include protocols (tcp, udp, igmp, eigrp, etc.), source and destination addresses and ports, and options that apply to the context of the traffic.

3.1. Practical Ruleset Development

This section concentrates on the basic commands and actions.

The basic command keywords are these:

- **enable/disable** - commands to disable or enable **ipfw** rule processing. The kernel module remains loaded - the effect is to suspend or resume rule processing. This is important to understand early as these commands function like an 'on/off' switch to firewall operation.
- **add** - adds a rule.
- **delete** - deletes a rule. The rule number must be specified - for example, **ipfw delete 1000**.
- **list** - lists the contents of the current ruleset. Even if there have been no rules added, the list command should always list out the default rule, by default, `65535 deny ip from any to any`.
- **show** - similar to the **list** command, **show** includes counters for each rule.
- **flush** - delete all the rules in the ruleset except for rules in set 31. (Sets are described later in this Chapter.) Since this is a command with enormous impact, a **Yes/No** prompt is issued before continuing.

The basic traffic flow action keywords are these:

- **allow** | **accept** | **pass** | **permit** - direct **ipfw** to allow a packet through this rule should the

packet match the rule body.

- **count** - increment a counter applied to a rule. No other processing is applied to the packet.
- **deny** | **drop** - do not allow a packet to pass through this rule should the packet match the rule body.
- **check-state** [:flowname | :any] - check if a dynamic rule already exists.
- **reset** - resets Network Address Translation tables.

The [ipfw\(8\)](#) man page has the complete list of action keywords, and describes each in detail.

This text also examines these keywords in the rules section:

- **prob** - assign a probability (a value between 0 and 1) to the rule action
- **set** - use a collection of rules
- **tag** and **untag** - apply an internal tag to a packet affected by the rule
- **log** and **logamount** - log keywords
- **reset** - send a TCP reset on a connection
- **tee** - cause packets to flow in multiple ways
- **unreach** - specify an action if a packet's destination is unreachable
- **setdscp** - set DiffServe parameters for outbound packets
- **skipto** - jump around a ruleset
- **divert** - pull packets into userspace for programmatic purposes
- **limit** - limit the number of active connections
- **call** and **return** - another way to jump around a ruleset
- **lookup tables** and the **lookup** keyword - value selection keywords

Most action keywords, such as **allow** or **deny**, determine traffic flow. It is important to become familiar with these actions as they will be used in almost every rule. In addition, carefully note what **ipfw** does *after* it matches a packet and applies an action - it either terminates its search, or it goes on to the next rule.

Other action keywords perform an activity that does not have any impact on traffic flow. For example, the **count** action simply updates counters that apply to a rule. It has no effect on traffic flow and **ipfw** continues processing with the next rule.

Recall that **ipfw** is a command line program that uses all the words on the command line as parameters. In developing rules, remember that certain constructs such as braces ({,}), brackets ([,]) and even parentheses themselves are all recognized by the shell and must be escaped with a

backslash `\`. The [ipfw\(8\)](#) man page has additional caveats on rule syntax.

To begin, shutdown all VMs and create the network architecture shown in [Chapter 2, Figure 1](#), including the required IP addresses.

```
# /bin/sh mkbr.sh reset bridge0 tap0 tap1 intf <--- use the appropriate interface on
the FreeBSD host
```

Start up the **firewall** VM, **external1** VM, and the **tmux** multiplexer for use with both VMs:

```
% sudo /bin/sh firewall.sh
% sudo /bin/sh external.sh
% /bin/sh swim.sh
```

On the **firewall** VM, load the **ipfw.ko** kernel module and start the **tserv.sh** service to listen for incoming connections.

Below is the smallest possible ruleset that permits the **external1** VM to make a TCP connection to the service running on the **firewall** VM.

```
# ipfw add 100 check-state
00100 check-state :default
# ipfw add 1000 allow tcp from any to me 5656 in via em0 setup keep-state
01000 allow tcp from any to me 5656 in via em0 setup keep-state :default
```

Test this ruleset immediately by again running **sh tserv.sh** on the **firewall** VM and **sh tcon.sh 5656** script on the **external1** VM as described in the previous chapter. The connection should succeed.

Rule 100 contains the **check-state** option. It checks to see if a connection is already established and a dynamic rule is in place. If so, any additional packets matching the dynamic rule would be passed. "Dynamic rules" are discussed shortly.

Rule 1000 contains the **add** command. This command inserts the requested rule into the **ipfw** ruleset where it can process packets against the specified actions in the rule body. The rule itself contains the **allow** keyword, which permits traffic to pass.

The rule also uses the **setup** and **keep-state** options to create a dynamic rule for the connection.

In a stateful firewall like **ipfw**, once a connection from an external host to an internal host is established, the firewall creates a dynamic rule permitting continued traffic along this path until the connection is reset.

Note that without the **check-state** keyword, no check for a dynamic rule is performed and without the **keep-state**, no creation of a dynamic rule is performed.

Consider this ruleset with just a single rule:

```
# ipfw add 1000 allow tcp from any to me 5656
```

This rule looks like it should work, **but it does not**. A TCP packet entering **ipfw** has no pre-existing dynamic rule. Further, the rule does not create one. And, because there is no corresponding rule for outbound traffic, no TCP 3-way handshake is ever completed. Note that a SYN packet *is* received by the firewall, but *not* by the destination service.

By adding the rule:

```
# ipfw add 2000 allow tcp from me to any
```

the TCP 3-way handshake **is** allowed to complete and the data is sent from the **external1** VM host to the **tserv.sh** process running on the **firewall** VM.

While this method works, it uses two rules instead of one. In this case, the better solution is to use the **setup**, **keep-state**, and **check-state** options early in the ruleset as shown in the original example in this section.

3.2. Dynamic Rules

So, what exactly are "dynamic rules"? The scripts being used close the TCP connection each time, so the dynamic rules are short lived, and cannot be easily examined.

To see dynamic rules in action, unload and reload the **ipfw** kernel module, and re-enter the original ruleset from the previous section.

Then manually set up an **ncat** listener on the **firewall** VM and send data with an **ncat** sender on the **external1** VM:

On the **firewall** VM, start up the listener service manually:

```
# ncat -l 203.0.113.50 5656
```

Then, on the **external1** VM, use **ncat** to connect to the service on the firewall and type a message:

```
# ncat 203.0.113.50 5656
hello there
^C
```

The message should appear on the console of the **firewall** VM. If it does not, ensure that the original rule from the previous section is active.

external1 VM host	Firewall
# ncat 203.0.113.50 5656 hello there	# ncat -l 203.0.113.50 5656 hello there

Figure 6. Manually Creating Traffic to Examine Dynamic Rules

The above figure shows the connection is open between the **external1** and **firewall** VMs.

While the connection is still open, run the following command on the **firewall** VM serial console:

```
# ipfw -d list
00100 check-state :default
01000 allow tcp from any to me in via em0 setup keep-state d:default
65535 deny ip from any to any
## Dynamic rules (1 152):
01000 STATE tcp 203.0.113.10 18618 <-> 203.0.113.50 5656 :default
#
```

Figure 7. Viewing Dynamic Rules

Output is similar to that in the above figure.

The **-d** option displays dynamic rules in addition to regular rules. The **-D** option displays just dynamic rules.

3.2.1. Notes on Rule Numbering

Each rule is assigned a rule number, even if one is not specified. The details for rule number handing are found in the [ipfw\(8\)](#) man page. Note that rules are assigned numbers in increments specified by the `sysctl net.inet.ip.fw.autoinc_step`.

```
# sysctl net.inet.ip.fw.autoinc_step
net.inet.ip.fw.autoinc_step: 100
```

Restart with a simple check-state rule and note that **ipfw** has assigned a number associated with the increment `sysctl` shown above:

```
Flush the ipfw ruleset first.

# ipfw -q flush
# ipfw add check-state
00000 check-state :default

# ipfw list
00100 check-state :default
65535 deny ip from any to any00000 check-state :default
```

ipfw has automatically assigned the rule number 100. While it can be convenient to have **ipfw** add a rule number automatically, it is best to *always* assign rule numbers yourself. This ensures a deliberate decision was made to put a rule in a specific place within the ruleset. With large rulesets this is critical. A rule automatically assigned by **ipfw** can be placed where it can have an

unexpected effect.

Consider this ruleset:

```
# ipfw list
00300 deny ip from any to 200.200.200.200
00400 deny ip from any to 200.200.200.201
00500 deny ip from any to 200.200.200.202
00600 deny ip from any to 200.200.200.203
00700 deny ip from any to 200.200.200.204
00800 deny ip from any to 200.200.200.205
00800 deny ip from any to 200.200.200.206
65535 deny ip from any to any
```

Having forgotten to add the check-state rule the firewall admin quickly adds it:

```
# ipfw add check-state
00000 check-state :default
```

resulting in the unintentional placement:

```
# ipfw list
00300 deny ip from any to 200.200.200.200
00400 deny ip from any to 200.200.200.200
00500 deny ip from any to 200.200.200.200
00600 deny ip from any to 200.200.200.200
00700 deny ip from any to 200.200.200.200
00800 deny ip from any to 200.200.200.200
00800 deny ip from any to 200.200.200.200
00900 check-state :default
65535 deny ip from any to any
```

Also, **ipfw** allows rules with the same rule number to be added to the ruleset, and it will keep track of the rules in the order they were entered. This is easy to forget when manually entering rules from the command line and using command line editing to change something simple like the last byte of an IP address.

It is important to remember that all such rules are affected by commands that operate on one or more lines, such as the **delete** command:

```
# ipfw add 100 check-state
00100 check-state :default
# ipfw add 1000 allow tcp from 203.0.113.10 to me 5656 setup keep-state
01000 allow tcp from 203.0.113.10 to me 5656 keep-state :default
# ipfw add 1000 allow tcp from 203.0.113.20 to me 5656 setup keep-state
01000 allow tcp from 203.0.113.20 to me 5656 keep-state :default
# ipfw add 1000 allow tcp from 203.0.113.30 to me 5656 setup keep-state
```

```

01000 allow tcp from 203.0.113.30 to me 5656 keep-state :default
# ipfw add 1000 allow tcp from 203.0.113.40 to me 5656 setup keep-state
01000 allow tcp from 203.0.113.40 to me 5656 keep-state :default
#
# ipfw list
00100 check-state :default
01000 allow tcp from 203.0.113.10 to me 5656 setup keep-state :default
01000 allow tcp from 203.0.113.20 to me 5656 setup keep-state :default
01000 allow tcp from 203.0.113.30 to me 5656 setup keep-state :default
01000 allow tcp from 203.0.113.40 to me 5656 setup keep-state :default
65535 deny ip from any to any
#
# ipfw delete 1000
#
# ipfw list
00100 check-state :default
65535 deny ip from any to any

```

The **delete** command can also process both ranges and lists of rules.

Consider the following ruleset:

```

# ipfw list
00100 check-state :default
01000 allow tcp from 203.0.113.10 to me 5656 setup keep-state :default
01100 allow tcp from 203.0.113.11 to me 5656 setup keep-state :default
01200 allow tcp from 203.0.113.12 to me 5656 setup keep-state :default
01300 allow tcp from 203.0.113.13 to me 5656 setup keep-state :default
02000 allow tcp from 203.0.113.20 to me 5656 setup keep-state :default
02100 allow tcp from 203.0.113.21 to me 5656 setup keep-state :default
02200 allow tcp from 203.0.113.22 to me 5656 setup keep-state :default
02300 allow tcp from 203.0.113.23 to me 5656 setup keep-state :default
03000 allow tcp from 203.0.113.30 to me 5656 setup keep-state :default
03100 allow tcp from 203.0.113.31 to me 5656 setup keep-state :default
03200 allow tcp from 203.0.113.32 to me 5656 setup keep-state :default
03300 allow tcp from 203.0.113.33 to me 5656 setup keep-state :default
04000 allow tcp from 203.0.113.40 to me 5656 setup keep-state :default
04100 allow tcp from 203.0.113.41 to me 5656 setup keep-state :default
04200 allow tcp from 203.0.113.42 to me 5656 setup keep-state :default
04300 allow tcp from 203.0.113.43 to me 5656 setup keep-state :default
04400 allow tcp from 203.0.113.44 to me 5656 setup keep-state :default
04500 allow tcp from 203.0.113.45 to me 5656 setup keep-state :default
04600 allow tcp from 203.0.113.46 to me 5656 setup keep-state :default
04700 allow tcp from 203.0.113.47 to me 5656 setup keep-state :default
04800 allow tcp from 203.0.113.48 to me 5656 setup keep-state :default
04900 allow tcp from 203.0.113.49 to me 5656 setup keep-state :default
65535 deny ip from any to any

```

A **range** is specified by two number separated by a dash: for example 5000-7350; whereas a **list** is a

space-separated collection of numbers on the command line.

The following command deletes rules from 1000 to 2999 and certain rules between 4000 and 5000:

```
# ipfw delete 1000-2999 4100 4300 4500 4700 4900
#
# ipfw list
00100 check-state :default
03000 allow tcp from 203.0.113.30 to me 5656 keep-state :default
03100 allow tcp from 203.0.113.31 to me 5656 keep-state :default
03200 allow tcp from 203.0.113.32 to me 5656 keep-state :default
03300 allow tcp from 203.0.113.33 to me 5656 keep-state :default
04000 allow tcp from 203.0.113.40 to me 5656 keep-state :default
04200 allow tcp from 203.0.113.42 to me 5656 keep-state :default
04400 allow tcp from 203.0.113.44 to me 5656 keep-state :default
04600 allow tcp from 203.0.113.46 to me 5656 keep-state :default
04800 allow tcp from 203.0.113.48 to me 5656 keep-state :default
65535 deny ip from any to any
```

Note that the **delete** command *will* operate on comma separated values, but the **delete** command will only remove the *first* value in a comma separated list, not the entire list. The command does not throw an error, but it does not delete all the lines requested.

```
# ipfw delete 3100,3200,3300
# echo $?
0          <--- No error found with the previous command.
#
# ipfw list
00100 check-state :default
03000 allow tcp from 203.0.113.30 to me 5656 keep-state :default
03200 allow tcp from 203.0.113.32 to me 5656 keep-state :default
03300 allow tcp from 203.0.113.33 to me 5656 keep-state :default
04000 allow tcp from 203.0.113.40 to me 5656 keep-state :default
04200 allow tcp from 203.0.113.42 to me 5656 keep-state :default
04400 allow tcp from 203.0.113.44 to me 5656 keep-state :default
04600 allow tcp from 203.0.113.46 to me 5656 keep-state :default
04800 allow tcp from 203.0.113.48 to me 5656 keep-state :default
65535 deny ip from any to any
```

The **show** command is similar to the **list** command but it also includes a packet count and byte count for each rule.

Stop any existing scripts on the **firewall** VM and run **sh userv3.sh**. Then create the following ruleset on the **firewall** VM:

```
# ipfw -q flush
#
# ipfw add 100 check-state
```

```

00100 check-state :default
# ipfw add 1000 allow udp from 203.0.113.10 to me 5656
01000 allow udp from 203.0.113.10 to me 5656
# ipfw add 2000 allow udp from 203.0.113.10 to me 5657
02000 allow udp from 203.0.113.10 to me 5657
# ipfw add 3000 allow udp from 203.0.113.10 to me 5658
03000 allow udp from 203.0.113.10 to me 5658
# ipfw add 4000 allow udp from 203.0.113.10 to me 5659
04000 allow udp from 203.0.113.10 to me 5659
#
# ipfw list
00100 check-state :default
01000 allow udp from 203.0.113.10 to me 5656
02000 allow udp from 203.0.113.10 to me 5657
03000 allow udp from 203.0.113.10 to me 5658
04000 allow udp from 203.0.113.10 to me 5659
65535 deny ip from any to any

```

Then, on the **external1** VM, run **sh uconr.sh 5656 1** script to send packets to ports 5656, 5657, and 5658, randomly:

```

# sh uconr.sh 5656 1
PORT1 = [5656]
SLEEPVAL = [1]
UDP packet from [203.0.113.10],[5656],[1]
UDP packet from [203.0.113.10],[5656],[2]
UDP packet from [203.0.113.10],[5658],[3]
UDP packet from [203.0.113.10],[5657],[4]
UDP packet from [203.0.113.10],[5659],[5]
UDP packet from [203.0.113.10],[5659],[6]
UDP packet from [203.0.113.10],[5659],[7]
UDP packet from [203.0.113.10],[5656],[8]
UDP packet from [203.0.113.10],[5658],[9]
UDP packet from [203.0.113.10],[5659],[10]
UDP packet from [203.0.113.10],[5658],[11]
UDP packet from [203.0.113.10],[5656],[12]
UDP packet from [203.0.113.10],[5656],[13]
UDP packet from [203.0.113.10],[5656],[14]
UDP packet from [203.0.113.10],[5659],[15]
UDP packet from [203.0.113.10],[5656],[16]
UDP packet from [203.0.113.10],[5657],[17]
UDP packet from [203.0.113.10],[5659],[18]
UDP packet from [203.0.113.10],[5659],[19]
UDP packet from [203.0.113.10],[5657],[20]
UDP packet from [203.0.113.10],[5659],[21]
^C

```

Running the **ipfw show** command outputs:

```
# ipfw show
00100 0 0 check-state :default
01000 7 494 allow udp from 203.0.113.10 to me 5656
02000 3 212 allow udp from 203.0.113.10 to me 5657
03000 3 211 allow udp from 203.0.113.10 to me 5658
04000 8 565 allow udp from 203.0.113.10 to me 5659
65535 11 897 deny ip from any to any
```

The output shows the number of packets and the number of bytes processed by each rule, including the default rule which may have processed many more packets.

This is a useful tool for debugging. Paired with the **zero** command which can clear counters with precise rule selection, it can show what rules are still processing a rule match.

The **zero** command takes a space separated list of rules (similar to the **delete** command) to clear counters. However, unlike the **delete** command, ranges (e.g 2000-3000) are not allowed.

```
# ipfw zero 2000 3000
#
# ipfw show
00100 0 0 check-state :default
01000 7 494 allow udp from 203.0.113.10 to me 5656
02000 0 0 allow udp from 203.0.113.10 to me 5657
03000 0 0 allow udp from 203.0.113.10 to me 5658
04000 8 565 allow udp from 203.0.113.10 to me 5659
65535 11 897 deny ip from any to any
```

Clearing all rule match counters can be done with **ipfw zero** with no parameters.

Clearing the default rule match counter can be done with **ipfw zero 65535**.

Counters are also a feature of rules that specify the **log** keyword. An example of this is shown below when discussing the **log** and **logamount** keywords.

3.3. Keywords

3.3.1. Protocols

Protocols are those defined by [IANA - the Internet Assigned Numbers Authority](https://www.iana.org) (<https://www.iana.org>) and are included in Unix systems in `/etc/protocols`. This file identifies what numbers are assigned to common (and some very obscure) protocols - **ip** (0), **tcp** (6), **udp** (17), **icmp** (1), and many others.

Source and destination protocols can be the conventional IP or IPv6 addresses. However, the [ipfw\(8\)](#) manual page has this more detailed explanation:

"The first part (proto from src to dst) is for backward compatibility with earlier versions of

FreeBSD. In modern FreeBSD any match pattern (including MAC headers, IP protocols, addresses and ports) can be specified in the options section."

The **ipfw** keywords for common protocols include:

ip4 | **ipv4** Matches IPv4 packets.

ip6 | **ipv6** Matches IPv6 packets.

ip | **all** Matches any IP packet.

The logical operator "**or**" can be used to combine multiple protocols where any one of them applies. The "{" and "}" braces can be used to group "**or**" conditions (known as "**or-blocks**"). Only one level of braces can be used. Braces must be escaped with a backslash '\' to prevent them from being interpreted directly by the command line shell:

```
# ipfw add 1100 deny \{ tcp or udp or eigrp or chaos \} from 1.2.3.4 to 5.6.7.8
01100 deny { tcp or udp or eigrp or chaos } from 1.2.3.4 to 5.6.7.8
```

Consider this ruleset in a shell script:

```
#!/bin/sh
```

```
ipfw add 5000 deny \{ icmp or ip or igmp or ggp or ipencap or st2 or tcp or cbt or egp or igp or bbn-
rcc or nvp or pup or argus or emcon or xnet or chaos or udp or mux or dcn or hmp or prm or xns-
idp or trunk-1 or trunk-2 or leaf-1 or leaf-2 or rdp or irtp or iso-tp4 or netblt or mfe-nsp or merit-inp
or dccp or 3pc or idpr or xtp or ddp or idpr-cmtp or tp++ or il or ipv6 or sdrp or ipv6-route or ipv6-
frag or idrp or rsvp or gre or dsr or bna or esp or ah or i-nlsp or swipecarp or narp or mobile or tlsp or
skip or ipv6-icmp or ipv6-nonxt or ipv6-opts or cftp or sat-expak or kryptolan or rvd or ippc or sat-
mon or visa or ipcw or cpnx or cphb or wsn or pvp or br-sat-mon or sun-nd or wb-mon or wb-expak
or iso-ip or vmtp or secure-vmtp or vines or ttp or nsfnet-igp or dgp or tcf or eigrp or ospf or sprite-
rpc or larp or mtp or ax.25 or ipip or micp or scc-sp or etherip or encap or gmtp or ifmp or pnni or
pim or aris or scps or qnx or a/n or ipcomp or snp or compaq-peer or ipx-in-ip or carp or pgm or
l2tp or ddx or iatp or stp or srp or uti or smp or sm or ptp or isis or fire or crtp or crudp or
sscopic or iplt or sps or pipe or sctp or fc or rsvp-e2e-ignore or mobility-header or udplite or
mpls-in-ip or manet or hip or shim6 or wesp or rohc or pfsync or divert \} from any to me
```

```
exit
```

Note that the above file is shown as one very long line and does not use the Unix line continuation convention.

This command will deny all traffic using all protocols defined in /etc/protocols. The above command will complete successfully. However, due to a bug in the "**or-block**" parser, the rule cannot have the

"ip" protocol first. Swapping the first two protocols - **icmp** and **ip** - the command throws an error.

For example,

```
# ipfw add 1000 deny \{ igmp or ip or eigrp \} from any to me
```

works Ok but

```
# ipfw add 1000 deny \{ ip or igmp or eigrp \} from any to me
ipfw: invalid OR block
```

fails.

The use of the logical "**and**" operator in a protocol block is an error. A packet can be in only one protocol at a time. However, the use of the logical "**not**" operator **is** permitted in front of a protocol identifier:

```
# ipfw add 1000 deny \{ icmp or not igmp \} from any to me
```

Careful consideration of all logical conditions is essential to correct operation of a ruleset.



In later versions of FreeBSD, the use of the protocol "**or-block**" is noted as deprecated in [ipfw\(8\)](#) but the operation may still complete successfully until the feature is removed completely.

3.3.2. Addresses

Source and destination addresses can be any of the following:

- **IPv4** or **IPv6** addresses
- **any** - matches any IP address.
- **me** - matches any IP address configured on an interface in the system.

Note that the interface does not have to have an IP or IPv6 address, **nor does it have to be up or even exist** at the time the rule is entered. Thus, be aware that a rule with keyword **me** may affect traffic on interfaces that are configured at a later time. Consider the following system interface list:

```
# ifconfig -a
em0: flags=8863<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> metric 0 mtu 1500

options=481209b<RXCSUM, TXCSUM, VLAN_MTU, VLAN_HWTAGGING, VLAN_HWCSUM, WOL_MAGIC, VLAN_HWFILTER, NOMAP>
ether 02:49:50:46:57:41
inet 203.0.113.50 netmask 0xffffffff broadcast 203.0.113.255
```

```
media: Ethernet autoselect (1000baseT <full-duplex>)
status: active
nd6 options=29<PERFORMNUD,IFDISABLED,AUTO_LINKLOCAL>
lo0: flags=8049<UP,LOOPBACK,RUNNING,MULTICAST> metric 0 mtu 16384
options=680003<RXCSUM, TXCSUM, LINKSTATE, RXCSUM_IPV6, TXCSUM_IPV6>
inet6 ::1 prefixlen 128
inet6 fe80::1%lo0 prefixlen 64 scopeid 0x3
inet 127.0.0.1 netmask 0xff000000
groups: lo
```

The following attempt to add a rule with a non-existent interface succeeds even though there is no `wlan0` interface defined:

```
# ipfw add 1000 deny tcp from 1.2.4.4 to me via wlan0
01000 deny tcp from 1.2.4.4 to me via wlan0
```

- **me6** - matches any IPv6 address similarly to the above me keyword.
- **table(name[,value])** - matches any IPv4 or IPv6 address for an entry in the named table.

Tables are discussed below in [Lookup Tables](#).

IPv4 and IPv6 addresses follow the usual conventions regarding address and mask or prefix length. Addresses can also be grouped into a list, similar to the capability discussed for protocols above.

```
# ipfw add 2000 allow tcp from \{ 2.3.4.5/32, 3.4.5.0/24, 10.0.0.0/8 \} to me
# ipfw add 2000 allow tcp from \{ 2607:fcc0:0:35::dd/64, 2608:abcd:0:2300::9eac/64 \}
to me
```

However, IPv4 and IPv6 addresses *cannot* be mixed in the same list. Use two different rules instead.

For sparse collections of addresses, consider the alternate form allowed by

```
addr-set: addr[/masklen]{list}
```

such as:

```
# ipfw add 1000 allow tcp from 1.2.3.0/24\{128,9,35-45,7\} to me
```

In this form, lists and increasing ranges are allowed. `ipfw` will consolidate overlapping ranges, and

will reorder the list in the display to show increasing addresses from left to right. Note that spaces are only allowed after commas between list elements, nowhere else.

This example does not work due to incorrect placement of spaces:

```
# ipfw add 1000 allow tcp from 1.2.4.0/24\{ 128,9,25-45,7-25 \} to me
ipfw: missing ``to''
```

This example works by correcting where spaces occur on the command line.

```
# ipfw add 1000 allow tcp from 1.2.4.0/24\{128,9,25-45,7-25\} to me
01000 allow tcp from 1.2.4.0/24{7-45,128} to me
```

As noted, **ipfw** will simplify, reorder, and display the list:

```
# ipfw list
01000 allow tcp from 1.2.4.0/24{7-45,128} to me
65535 deny ip from any to any
```

Note that ranges *must* be defined as increasing. Also, as noted in [ipfw\(8\)](#), there is **no** support for sets of IPv6 addresses.

3.3.3. Ports

Ports may be specified by number or by service name. Service names, also under the provenance of [IANA](#), are found in `/etc/services` on Unix systems.

Ports can be specified as individual items, lists, or ranges. Typically ports are used to determine a destination service and so apply to the destination address (although specifying source ports is also permitted):

```
# ipfw add 1000 allow tcp from 1.2.3.4 to me daytime
01000 allow tcp from 1.2.3.4 to me 13

# ipfw add 2000 allow tcp from 2.3.4.5 to me ssh, telnet, smtp
02000 allow tcp from 2.3.4.5 to me 22,23,25

# ipfw add 3000 allow tcp from 3.4.5.6 to me auditd-domain
03000 allow tcp from 3.4.5.6 to me 48-53

# This example uses source and destination ports.
# ipfw add 4000 allow tcp from 7.8.9.10 3030 to me ssh
04000 allow tcp from 7.8.9.10 3030 to me 22
```

General Notes on Port Ranges

1. A range such as that shown above (`auditd-domain`) may accidentally include ports not wanted. In this case, ports `tacacs` (49), `re-mail-ck` (50), `la-maint` (51), and `xns-time` (52) would be included. Therefore, always check actual port numbers when using named ranges for ports.
2. Some service names include the dash character '-' as part of the name, as in the point above. In these cases a *double backslash*, is required, one for the shell and one for `ipfw`:

```
# ipfw add 4000 allow tcp from 4.5.6.7 to me ftp, ftp\\-data
04000 allow tcp from 4.5.6.7 to me 21,20
```

3. Some applications require a range of source and destination ports in both directions. This is easy to accomplish with ranges and a `keep-state` rule:

```
# ipfw add 1000 allow tcp from 203.0.113.10 5200-5205 to me 5656-5658 keep-state
01000 allow tcp from 203.0.113.10 5200-5205 to me 5656-5658 keep-state :default
#
# ipfw list
01000 allow tcp from 203.0.113.10 5200-5205 to me 5656-5658 keep-state :default
65535 deny ip from any to any
```

4. The syntactic sugar provided by the match keywords `dst-port` and `src-port` are both part of the match section of the rule:

```
# ipfw add 1000 allow tcp from 203.0.113.10 to me src-port 3030 dst-port 1010
01000 allow tcp from 203.0.113.10 3030 to me 1010
```

To test common ports in both directions, manually connect using `ncat`, setting up the source and destination ports as needed:

```
# ipfw -q flush
# ipfw add 1000 allow tcp from 203.0.113.10 to me src-port 5200=5205 dst-port 5656-
5658 keep-state
01000 allow tcp from 203.0.113.10 5200-5205 to me 5656-5658 keep-state :default
# ipfw list
01000 allow tcp from 203.0.113.10 5200-5205 to me 5656-5658 keep-state :default
65535 deny ip from any to any
```

Note: the transmission and reception lines have been aligned on each side.

External1 VM host	Firewall VM host
<pre># echo "hello to dstport 5656 from srcport 5200" ncat -p 5200 203.0.113.50 5656 # echo "hello to dstport 5657 from srcport 5200" ncat -p 5200 203.0.113.50 5657 # echo "hello to dstport 5658 from srcport 5200" ncat -p 5200 203.0.113.50 5658 # echo "hello to dstport 5659 from srcport 5200" ncat -p 5200 203.0.113.50 5659 Ncat: TIMEOUT. # # echo "hello to dstport 5656 from srcport 5204" ncat -p 5204 203.0.113.50 5656 # echo "hello to dstport 5656 from srcport 5205" ncat -p 5205 203.0.113.50 5656 # echo "hello to dstport 5656 from srcport 5206" ncat -p 5206 203.0.113.50 5656 Ncat: TIMEOUT. ^C #</pre>	<pre># sh tserv3.sh Starting TCP listeners on [5656],[5657],[5658] hello to dstport 5656 from srcport 5200 hello to dstport 5657 from srcport 5200 hello to dstport 5658 from srcport 5200 hello to dstport 5656 from srcport 5204 hello to dstport 5656 from srcport 5205 ^C #</pre>

Figure 8. Manually Testing Common Ports in Both Directions

Note that no connection was achieved when the destination port was out of bounds (5659) and when the source port was out of bounds (5206).

3.3.4. Prob

The **prob** keyword is used to assign a chance, that is, a probability (a floating point value between 0 and 1), that an incoming packet will be matched. If the chance is successful, the corresponding rule performs the required action. If the chance is not successful, the rule is not matched and rule processing continues to the next rule.

To test this keyword, use "**sh ucont.sh 5656 1**" script on the **external1** VM's side to repeatedly send a UDP packet to the **firewall** VM, who is listening on UDP port 5656. Using the **prob** keyword, set a probability of .5 (a 50% chance) that the packet will be matched. The action is to let the packet pass to the service, which just prints the contents of the packet.

As shown below, there were 24 out of 50 packets received, very close to 50% for such a small sample:

```
# ipfw -q flush
#
# ipfw add 3000 prob 0.5 allow udp from any to me 5656
03000 prob 0.500000 allow udp from any to me 5656
#
# ipfw list
03000 prob 0.500000 allow udp from any to me 5656
65535 deny ip from any to any
#
# sh userv.sh 5656
PORT1 = [5656]
Starting UDP listener on [203.0.113.50],[5656]
UDP packet from [203.0.113.10],[5656],[5]
UDP packet from [203.0.113.10],[5656],[6]
UDP packet from [203.0.113.10],[5656],[7]
```

```
UDP packet from [203.0.113.10],[5656],[10]
UDP packet from [203.0.113.10],[5656],[11]
UDP packet from [203.0.113.10],[5656],[13]
UDP packet from [203.0.113.10],[5656],[14]
UDP packet from [203.0.113.10],[5656],[16]
UDP packet from [203.0.113.10],[5656],[19]
UDP packet from [203.0.113.10],[5656],[20]
UDP packet from [203.0.113.10],[5656],[22]
UDP packet from [203.0.113.10],[5656],[28]
UDP packet from [203.0.113.10],[5656],[29]
UDP packet from [203.0.113.10],[5656],[32]
UDP packet from [203.0.113.10],[5656],[33]
UDP packet from [203.0.113.10],[5656],[34]
UDP packet from [203.0.113.10],[5656],[35]
UDP packet from [203.0.113.10],[5656],[37]
UDP packet from [203.0.113.10],[5656],[38]
UDP packet from [203.0.113.10],[5656],[39]
UDP packet from [203.0.113.10],[5656],[41]
UDP packet from [203.0.113.10],[5656],[43]
UDP packet from [203.0.113.10],[5656],[45]
UDP packet from [203.0.113.10],[5656],[49]
^C#
```

3.3.5. Sets

Firewall rules can be grouped into different **sets** which can be switched atomically. Why use this feature? Consider a datacenter with two sets of identical servers on separate networks. One set must be taken down for maintenance. But first, traffic must be transferred to the other set of servers. Using **sets** is a practical solution for this problem.

Sets are useful, but they do come with some caveats which are described throughout this section.

The default **set** is **set 0**. To begin, create rules in **set 0** and **set 1** with slight differences between the two. Note that this example also shows the use of the **ipfw** comment feature - allowing comments on a per-rule basis.

```
# ipfw -q flush
#
# ipfw add 1000 set 0 check-state
01000 check-state :default
#
# ipfw add 1100 set 0 allow tcp from any to me 5656 setup keep-state // 5656 only
01100 allow tcp from any to me 5656 setup keep-state :default // 5656 only
#
# ipfw add 2000 set 1 check-state
02000 check-state :default
#
# ipfw add 2100 set 1 allow tcp from any to me 5657 setup keep-state // 5657 only
02100 allow tcp from any to me 5657 setup keep-state :default // 5657 only
```

```
#
# ipfw set show
enable 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
30
```

The above **ipfw set show** command lists all enabled sets, here showing both 0 and 1 as enabled.

From the **external1** VM, commence communications using the **tcon.sh** script as shown below. The first two communications ([1], [2]) are to port 5656, the next two communications are to port 5657 ([3],[4]). The firewall host shows all four communications received:

External1 VM host	Firewall VM host
<pre># sh tcon.sh 5656 TCP connection from [203.0.113.50],[5656],[1] ncat [2] ready. Enter a valid PORTNUM: TCP connection from [203.0.113.50],[5656],[2] ncat [3] ready. Enter a valid PORTNUM: 5657 TCP connection from [203.0.113.50],[5657],[3] ncat [4] ready. Enter a valid PORTNUM: TCP connection from [203.0.113.50],[5657],[4] ncat [5] ready. Enter a valid PORTNUM: 5656 TCP connection from [203.0.113.50],[5656],[5] Ncat: TIMEOUT. TCP connection [203.0.113.10],[5656],[5] FAILED ncat [6] ready. Enter a valid PORTNUM: TCP connection from [203.0.113.50],[5656],[6] Ncat: TIMEOUT. TCP connection [203.0.113.10],[5656],[6] FAILED ncat [7] ready. Enter a valid PORTNUM: 5657 TCP connection from [203.0.113.50],[5657],[7] ncat [8] ready. Enter a valid PORTNUM: TCP connection from [203.0.113.50],[5657],[8] ncat [9] ready. Enter a valid PORTNUM:</pre>	<pre># sh tserv3.sh Starting TCP listeners on [5656],[5657],[5658] TCP connection from[203.0.113.10],[5656],[1] TCP connection from[203.0.113.10],[5656],[2] TCP connection from[203.0.113.10],[5657],[3] TCP connection from[203.0.113.10],[5657],[4] TCP connection from[203.0.113.10],[5657],[7] TCP connection from[203.0.113.10],[5657],[8]</pre>

Figure 9. Use of Sets

Right before communication 5, the firewall host admin disabled **set 0**, (**# ipfw set disable 0**) effectively blocking access to port 5656. Disabling **set 0**, effectively removes the rules for port 5656 and so the communications [5] and [6] fail. The **external1** VM goes back to port 5657 for communications [7] and [8], which are successful.

Notice that when listing the firewall ruleset with just **ipfw list**, only the **sets** that are enabled actually show up. To make sure which **sets** are enabled / disabled, use the **-S** flag on the **ipfw** command as shown below.

```
# ipfw set enable 0
#
# ipfw set show
enable 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
30
#
# ipfw -S list
01000 set 0 check-state :default
01100 set 0 allow tcp from any to me 5656 setup keep-state :default // 5656 only
02000 set 1 check-state :default
```

```

02100 set 1 allow tcp from any to me 5657 setup keep-state :default // 5657 only
65535 set 31 deny ip from any to any
#
# ipfw set disable 0
#
# ipfw set show
disable 0 enable 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
27 28 29 30
#
# ipfw -S list
# DISABLED 01000 set 0 check-state :default
# DISABLED 01100 set 0 allow tcp from any to me 5656 setup keep-state :default // 5656
only
02000 set 1 check-state :default
02100 set 1 allow tcp from any to me 5657 setup keep-state :default // 5657 only
65535 set 31 deny ip from any to any
#

```

To atomically change from one set to the other use the **ipfw set swap** command:

```
# ipfw set swap 0 1
```

What actually happens is that *the rules in the sets get swapped*; that is, all the rules in **set 0** get put in **set 1** and all the rules in **set 1** get put in **set 0**.

```

#
# ipfw -S list
# DISABLED 01000 set 0 check-state :default
# DISABLED 01100 set 0 allow tcp from any to me 5656 setup keep-state :default // 5656
only
02000 set 1 check-state :default
02100 set 1 allow tcp from any to me 5657 setup keep-state :default // 5657 only
65535 set 31 deny ip from any to any
#
# ipfw set swap 0 1
#
# ipfw -S list
01000 set 1 check-state :default
01100 set 1 allow tcp from any to me 5656 setup keep-state :default // 5656 only
# DISABLED 02000 set 0 check-state :default
# DISABLED 02100 set 0 allow tcp from any to me 5657 setup keep-state :default // 5657
only
65535 set 31 deny ip from any to any

```

Note carefully that when swapping **sets** where one of the **sets** is disabled, **the set number is still disabled after the swap**, even though the rules are now different. This can lead to unexpected consequences such as the following:

```

# ipfw -S list
01000 set 0 check-state :default
01100 set 0 allow tcp from any to me 5656 setup keep-state :default // 5656 only
02000 set 1 check-state :default
02100 set 1 allow tcp from any to me 5660 setup keep-state :default // 5660 only
65535 set 31 deny ip from any to any
#
# ipfw set disable 0
#
# ipfw -S list
# DISABLED 01000 set 0 check-state :default
# DISABLED 01100 set 0 allow tcp from any to me 5656 setup keep-state :default // 5656
only
02000 set 1 check-state :default
02100 set 1 allow tcp from any to me 5660 setup keep-state :default // 5660 only
65535 set 31 deny ip from any to any
#
# ipfw set swap 0 1
#
# ipfw -S list
01000 set 1 check-state :default
01100 set 1 allow tcp from any to me 5656 setup keep-state :default // 5656 only
# DISABLED 02000 set 0 check-state :default
# DISABLED 02100 set 0 allow tcp from any to me 5660 setup keep-state :default // 5660
only
65535 set 31 deny ip from any to any

```

Here **set 0** is disabled, and then swapped. After the swap, **set 0 is still disabled**, though the rules have changed.

Note that all sets are initially enabled. When a set is disabled, say **set 3**, all other sets are still active, even if no rule references them. Sets are analogous to the pieces on the back row of a chess board. If a knight or a bishop is removed, that one piece is not able to play, but all the others are able to play.

set 31 cannot be deleted or changed. It can however partially participate in a swap.

```
# ipfw set swap 1 31
```

This swap will complete successfully (return code 0), but the effect is not the same as the swaps above. The default rule in **set 31** is not swapped, but the set number for the other rules (rules in **set 1**) are set to 31:

```

# ipfw -S list
01000 set 1 check-state :default
01100 set 1 allow tcp from 1.2.3.4 to me 8080
01200 set 1 allow tcp from 1.2.3.4 to me 8081
65535 set 31 deny ip from any to any

```

```
#
# ipfw set swap 1 31
#
# ipfw -S list
01000 set 31 check-state :default
01100 set 31 allow tcp from 1.2.3.4 to me 8080
01200 set 31 allow tcp from 1.2.3.4 to me 8081
65535 set 31 deny ip from any to any
```

As noted in the [ipfw\(8\)](#) manual page, rules in **set 31** cannot be flushed. There are now 4 rules in **set 31**:

```
# ipfw -f flush
Flushed all rules.
#
# ipfw -S list
01000 set 31 check-state :default
01100 set 31 allow tcp from 1.2.3.4 to me 8080
01200 set 31 allow tcp from 1.2.3.4 to me 8081
65535 set 31 deny ip from any to any
```

While **ipfw flush** did not clean out rules in **set 31**, the command **ipfw delete set 31** will clean out all but the default rule:

```
# ipfw delete set 31
#
# ipfw -S list
65535 set 31 deny ip from any to any
```

Further, note that any **set** that is disabled, remains disabled after a flush. Thus, when disabling a **set** and then flushing the entire ruleset, any rules added back into the disabled **set** number **will still be disabled**. This includes **set 0**, the default set.

Consider the following:

```
# ipfw list
01000 check-state :default
01100 allow tcp from any to me 1111
02000 allow tcp from any to me 2222
03000 allow tcp from any to me 3333
65535 deny ip from any to any
#
# ipfw set disable 0
#
# ipfw -S list
# DISABLED 01000 set 0 check-state :default
# DISABLED 01100 set 0 allow tcp from any to me 1111
```

```

# DISABLED 02000 set 0 allow tcp from any to me 2222
# DISABLED 03000 set 0 allow tcp from any to me 3333
65535 set 31 deny ip from any to any
#
# ipfw -f flush
Flushed all rules.
#
# ipfw add 100 check-state
00100 check-state :default
# ipfw add 200 allow tcp from any to me 5555
00200 allow tcp from any to me 5555
# ipfw add 300 allow tcp from any to me 6666
00300 allow tcp from any to me 6666
# ipfw add 400 allow tcp from any to me 7777
00400 allow tcp from any to me 7777
#
# ipfw -S list
# DISABLED 00100 set 0 check-state :default
# DISABLED 00200 set 0 allow tcp from any to me 5555
# DISABLED 00300 set 0 allow tcp from any to me 6666
# DISABLED 00400 set 0 allow tcp from any to me 7777
65535 set 31 deny ip from any to any
#

```

Because **set 0** was disabled before the flush, the flush has no effect on the enable/disable state of that **set**.

Note that it is even possible to disable sets of rules *that do not yet exist*:

```

# kldload ipfw
ipfw2 (+ipv6) initialized, divert loadable, nat loadable, default to deny, logging
disabled
#
# ipfw -S list
65535 set 31 deny ip from any to any
#
# ipfw set show
enable 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
30
#
# ipfw set disable 4 5 6 7 8 9
#
# ipfw set show
disable 4 5 6 7 8 9 enable 0 1 2 3 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
27 28 29 30
#

```

Using **sets** can be very helpful, as long as their properties and limitations are clearly understood.

3.3.6. Tags

Tags allow for marking incoming packets in such a way that later rules can be applied based on the **tag**.

For a simple example, consider tagging incoming packets from different networks. Later rules determine if the tagged packets are allowed or denied:

```
# ipfw add 100 check-state
00100 check-state :default
#
# ipfw add 1000 count tag 10 tcp from 172.16.200.0/24 to me 5656
01000 count tag 10 tcp from 172.16.200.0/24 to me 5656
#
# ipfw add 1100 count tag 20 tcp from 172.16.225.0/24 to me 5656
01100 count tag 20 tcp from 172.16.225.0/24 to me 5656
#
# ipfw add 1200 count tag 30 tcp from 203.0.113.0/24 to me 5656
01200 count tag 30 tcp from 203.0.113.0/24 to me 5656
#
# ipfw add 3000 allow tcp from any to me tagged 30 setup keep-state
03000 allow tcp from any to me tagged 30 setup keep-state :default
#
# ipfw add 4000 deny tcp from any to me tagged 10,20
04000 deny tcp from any to me tagged 10,20
#
```

Test this by using `ncat(1)` to set its own source address. To do this, first setup two alias address on the `em0` interface on the **external1** VM:

```
# ifconfig em0 172.16.200.10/24 alias
# ifconfig em0 172.16.225.10/24 alias
#
# ifconfig em0
em0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> metric 0 mtu 1500
options=81209b<RXCSUM, TXCSUM, VLAN_MTU, VLAN_HWTAGGING, VLAN_HWCSUM, WOL_MAGIC, VLAN_HWFILTER>
ether 02:49:50:46:57:10
inet 203.0.113.10 netmask 0xfffff00 broadcast 203.0.113.255
inet 172.16.200.10 netmask 0xfffff00 broadcast 172.16.200.255
inet 172.16.225.10 netmask 0xfffff00 broadcast 172.16.225.255
media: Ethernet autoselect (1000baseT <full-duplex>)
status: active
nd6 options=29<PERFORMNUD,IFDISABLED,AUTO_LINKLOCAL>
```


External1 VM host	Firewall VM host
<pre># # ncat --source 172.16.200.10 203.0.113.50 5656 Ncat: TIMEOUT. # ncat --source 172.16.225.10 203.0.113.50 5656 Ncat: TIMEOUT. # ncat --source 203.0.113.10 203.0.113.50 5656 hello from 203.0.113.10 ^C #</pre>	<pre># # sh tserv.sh 5656 Starting TCP listener on [5656] hello from 203.0.113.10 ^C #</pre>

Figure 10. Use of Tags

Because of the rule tagging in this ruleset, only traffic tagged with value "30" is allowed to pass.

Tags, combined with lookup tables allow for powerful policy based network access.

3.3.7. Logging

ipfw supports two methods of logging:

3.3.7.1. Method 1 – using **ipfw0**, the IPFW pseudointerface

```
# kldload ipfw
# ifconfig ipfw0 create
```

Note that the **ipfw.ko** kernel module must be loaded before creating the **ipfw0** interface. Also, note that if **ipfw.ko** is unloaded, the interface is destroyed and is no longer available.

Why use the **ipfw0** interface?

It is possible to read logs in real time with programs such as [tcpdump\(1\)](#), [wireshark\(1\)](#), or other network monitoring programs. This includes viewing the entire packet.

An example is given further below.

3.3.7.2. Method 2 – use **syslogd**

Setting the `sysctl` variable `net.inet.ip.fw.verbose = 1` will instruct the firewall to log packets to [syslogd\(8\)](#) even when the **ipfw0** interface exists. `syslogd` must be configured via `/etc/syslog.conf`. **ipfw** packets will be logged with a `LOG_SECURITY` facility. The logging limit is configurable via `net.inet.ip.fw.verbose_limit`, which is set to 0 (unlimited) by default.

Why use the **syslogd** interface?

Of the two methods, it is the only one that processes count actions, and is also the only one that prints rule numbers with the log entry.

To test logging, create a rule with the **log** keyword:

```
# ipfw add 1000 allow log udp from 203.0.113.10 to me 5656
```

Counters are also a feature of rules that specify the **log** keyword. If rules in a ruleset are edited to add the **log** keyword, matches for all rules will be included in the log entries with associated counts.

Create a ruleset using the **log** keyword that looks like this:

```
01000 allow log udp from 203.0.113.10 to me 5656
02000 allow log udp from 203.0.113.10 to me 5657
03000 allow log udp from 203.0.113.10 to me 5658
04000 allow log udp from 203.0.113.10 to me 5659
65535 deny ip from any to any
```

3.3.7.3. Using Method 1

To use Method 2, first set the required **sysctl** variable:

```
# sysctl net.inet.ip.fw.verbose=0
```

Then, capture/view logs with **tcpdump**:

```
# tcpdump -i ipfw0 -X -v
```

Experiment with the **tcpdump -v**, **-vv**, and **-vvv** options, which gives increasingly more verbose output. Consult [tcpdump\(1\)](#) for details.



Be sure to remove the address aliases added to the **em0** interface on the **external1** VM from the previous Section on [Tags](#).

The example below examines traffic with the **ucont.sh** script on the **external1** VM and the **usersv3.sh** script on the **firewall** VM.

External1 VM host	Firewall VM host
<pre># sh ucont.sh 5656 1 PORT1 = [5656] SLEEPVAL = [1] UDP packet from [203.0.113.10], [5656], [1] UDP packet from [203.0.113.10], [5656], [2] UDP packet from [203.0.113.10], [5656], [3] UDP packet from [203.0.113.10], [5656], [4] ^C</pre>	<pre># sh usersv3.sh Starting UDP listeners on [5656],[5657],[5658] UDP packet from [203.0.113.10], [5656], [1] UDP packet from [203.0.113.10], [5656], [2] UDP packet from [203.0.113.10], [5656], [3] UDP packet from [203.0.113.10], [5656], [4] ^C#</pre>

Figure 11. UDP Traffic From External1 to Firewall

Logged traffic from the above communication appears on the log device **ipfw0**:

```
#
# tcpdump -i ipfw0 -X -vvv
tcpdump: WARNING: ipfw0: That device doesn't support promiscuous mode
(BIOCPRMISC: Invalid argument)
tcpdump: listening on ipfw0, link-type EN10MB (Ethernet), capture size 262144 bytes
```

```

20:50:44.259127 IP (tos 0x0, ttl 64, id 61929, offset 0, flags [none], proto UDP (17),
length 70)
  203.0.113.10.27337 > 203.0.113.50.5656: [udp sum ok] UDP, length 42
    0x0000:  4500 0046 f1e9 0000 4011 1c61 ac10 0a0a  E..F....@..a....
    0x0010:  ac10 0a32 6ac9 1618 0032 00ff 5544 5020  ...2j....2..UDP.
    0x0020:  7061 636b 6574 2066 726f 6d20 5b31 3732  packet.from.[172
    0x0030:  2e31 362e 3130 2e31 305d 2c5b 3536 3536  .16.10.10],[5656
    0x0040:  5d2c 5b31 5d0a                                ],[1].
20:50:44.581025 IP (tos 0x0, ttl 64, id 61930, offset 0, flags [none], proto UDP (17),
length 70)
  203.0.113.10.41914 > 203.0.113.50.5656: [udp sum ok] UDP, length 42
    0x0000:  4500 0046 f1ea 0000 4011 1c60 ac10 0a0a  E..F....@..`....
    0x0010:  ac10 0a32 a3ba 1618 0032 c80c 5544 5020  ...2.....2..UDP.
    0x0020:  7061 636b 6574 2066 726f 6d20 5b31 3732  packet.from.[172
    0x0030:  2e31 362e 3130 2e31 305d 2c5b 3536 3536  .16.10.10],[5656
    0x0040:  5d2c 5b32 5d0a                                ],[2].
20:50:45.960845 IP (tos 0x0, ttl 64, id 61931, offset 0, flags [none], proto UDP (17),
length 70)
  203.0.113.10.33126 > 203.0.113.50.5656: [udp sum ok] UDP, length 42
    0x0000:  4500 0046 f1eb 0000 4011 1c5f ac10 0a0a  E..F....@.._....
    0x0010:  ac10 0a32 8166 1618 0032 ea5f 5544 5020  ...2.f...2..UDP.
    0x0020:  7061 636b 6574 2066 726f 6d20 5b31 3732  packet.from.[172
    0x0030:  2e31 362e 3130 2e31 305d 2c5b 3536 3536  .16.10.10],[5656
    0x0040:  5d2c 5b33 5d0a                                ],[3].
^C

```

3.3.7.4. Using Method 2

To use Method 2, first set the required `sysctl` variable:

```
# sysctl net.inet.ip.fw.verbose=1
```

Next, examine `/etc/syslog.conf` to see if there is already a facility and level for security listed. In modern versions of FreeBSD it is common to see:

```
security.*                /var/log/security
```

`ipfw` creates logs with the `LOG_SECURITY` facility, and `INFO` and `DEBUG` levels, and sends output to the file `/var/log/security` in this case.

If this entry exists in `/etc/syslog.conf`, the system is all set. Otherwise read through the below and set up an entry for an `ipfw` logfile.

To log to `syslogd(8)`, add the following line to the end of `/etc/syslog.conf`:

```
security.*                /var/log/security
```

(FreeBSD /etc/syslog.conf allows tabs or spaces to be used in the file.)

Create the logfile with

```
# touch /var/log/security
```

then send a **HANGUP** signal to the **syslogd** daemon:

```
# kill -HUP <pid of syslogd>
```

Re-run the above example and Use **tail -f** to see logs in real time.

Note that **ipfw** only logs *matched rules* with this method:

```
# tail -f /var/log/security
Apr  3 14:50:12 firewall newsyslog[401]: logfile first created
Apr  9 21:05:13 firewall kernel: ipfw: 1000 Accept UDP 203.0.113.10:59203
203.0.113.50:5656 in via em0
Apr  9 21:05:15 firewall kernel: ipfw: 1000 Accept UDP 203.0.113.10:12401
203.0.113.50:5656 in via em0
Apr  9 21:05:16 firewall kernel: ipfw: 1000 Accept UDP 203.0.113.10:45319
203.0.113.50:5656 in via em0
^C
```

The **log** entry includes the date, time, host, service, and rule number (1000 above) to make it easy to track which rule is being matched.

General Notes on logging

Consider the following ruleset:

```
# ipfw list
00100 check-state :default
01000 allow tcp from 203.0.113.100 to me setup keep-state :default
02000 allow icmp from 203.0.113.100 to me
02100 allow icmp from me to 203.0.113.100
03000 allow log udp from 203.0.113.10 to me 5656
04000 allow log logamount 20 udp from 203.0.113.10 to me 5657
05000 allow log logamount 20 udp from 203.0.113.10 to me 5658
65535 deny ip from any to any
```

When using Method 2 (syslog) on a quiet system, notice that the entries do not appear right away when reading the security log file in real time (for example, **tail -f /var/log/security**). This is because **syslogd** will buffer identical lines and output a notification only occasionally as in the below example:

```
Mar 28 22:30:01 firewall kernel: ipfw: 3000 Accept UDP 203.0.113.10:27519
203.0.113.50:5656 in via em0
Mar 28 22:30:03 firewall syslogd: last message repeated 4 times
Mar 28 22:32:31 firewall syslogd: last message repeated 31 times
```

Also, Method 1 (using the **ipfw0** interface) and Method 2 (syslog) are *mutually exclusive*. It is not possible to have both active at the same time. If `net.inet.ip.fw.verbose=0`, the output will be to the **ipfw0** interface; if the value is 1, the log output will be to **syslog**.

logamount values in rules only apply to **Method 2 - syslog**. They have no effect on limiting the number of packets sent out the **ipfw0** interface.

In **Method 2 - syslog**, when the log limit is reached, **ipfw** will send a notification similar to the following into the designated security logging file (default: `/var/log/security`):

```
Mar 28 23:00:10 firewall kernel: ipfw: 5000 Accept UDP 203.0.113.10:63367
203.0.113.50:5658 in via em0
Mar 28 23:00:11 firewall kernel: ipfw: 5000 Accept UDP 203.0.113.10:30909
203.0.113.50:5658 in via em0
Mar 28 23:00:11 firewall kernel: ipfw: limit 20 reached on entry 5000
```

And, at the same time, it also conveniently sends the same message to `/var/log/messages`, the standard FreeBSD logfile:

```
Mar 28 23:00:11 firewall kernel: ipfw: limit 20 reached on entry 5000
```

After that notification is sent, no more syslog entries will be sent from the matching rule until the log counters are reset with:

```
# ipfw resetlog <rule number>
```

When the **resetlog** command is entered, **ipfw** will send a reset notification to syslog:

```
Mar 28 23:04:51 firewall kernel: ipfw: logging count reset.
```

Unfortunately, as of FreeBSD version 14.1, it does not say *which* rule the count was reset for. Presumably, the firewall admin should know which rule since they just entered the command, at least if there is only one admin. In any case, it is a good idea to keep track of that manually when working with many rules that include the **log** keyword.

If issuing an **ipfw resetlog** command *without* specifying a rule number, **all** counters in all rules are reset and **ipfw** sends the following notification:

```
Mar 28 23:08:52 firewall kernel: ipfw: All logging counts reset.
```

Finally, note that the sysctl variable `net.inet.ip.fw.verbose_limit` provides a "default limit" if one is not specified with the **logamount** keyword in the ruleset.

Consider this scenario:

```
# sysctl net.inet.ip.fw.verbose_limit
net.inet.ip.fw.verbose_limit: 5 <---- The limit is preset to 5

# ipfw list
65535 deny ip from any to any
```

As new rules are added, **ipfw** will apply any **logamount** value it finds in the body of a rule. If a rule being entered does not have a **logamount** entry, the value defaults to the current `net.inet.ip.fw.verbose_limit` amount.

```
# ipfw add 100 check-state
00100 check-state :default
#
# ipfw add 3000 allow log udp from 203.0.113.10 to me 5656
03000 allow log logamount 5 udp from 203.0.113.10 to me 5656
#
# ipfw add 4000 allow log logamount 20 udp from 203.0.113.10 to me 5657
04000 allow log logamount 20 udp from 203.0.113.10 to me 5657
#
# ipfw list
00100 check-state :default
03000 allow log logamount 5 udp from 203.0.113.10 to me 5656
04000 allow log logamount 20 udp from 203.0.113.10 to me 5657
65535 deny ip from any to any
```

If the sysctl for `net.inet.ip.fw.verbose_limit` is changed *after* the rule is entered, it has *no effect*:

```
# sysctl net.inet.ip.fw.verbose_limit=3
net.inet.ip.fw.verbose_limit: 5 -> 3
```

and later in `/var/log/messages`

```
Mar 29 11:07:02 firewall kernel: ipfw: limit 5 reached on entry 3000
...
Mar 29 11:07:24 firewall kernel: ipfw: limit 20 reached on entry 4000
```

3.3.8. Reset

The **reset** keyword sends an immediate TCP reset on a rule match containing that keyword. This immediately shuts down any TCP connection from the source matching the rule. Create a new ruleset as follows:

```
# ipfw list
00100 check-state :default
01000 allow log tcp from 203.0.113.10 to me 5656 setup keep-state :default
02000 reset log tcp from 203.0.113.10 to me 5657
03000 reset log udp from 203.0.113.10 to me 5658
65535 deny ip from any to any
```

Test rule 2000 by executing **sh tcon.sh 5657** from the **external1** VM. Note that the corresponding script, "sh tserv.sh" does not even have to be running.

The syslog view of a TCP **reset** rule match looks like this:

```
Apr 9 21:44:49 firewall kernel: ipfw: 1000 Accept TCP 203.0.113.50:5656
203.0.113.10:28218 out via em0
Apr 9 21:44:49 firewall syslogd: last message repeated 1 times
Apr 9 21:44:49 firewall kernel: ipfw: 1000 Accept TCP 203.0.113.10:28218
203.0.113.50:5656 in via em0
Apr 9 21:45:01 firewall kernel: ipfw: 2000 Reset TCP 203.0.113.10:12998
203.0.113.50:5657 in via em0
Apr 9 21:45:07 firewall kernel: ipfw: 2000 Reset TCP 203.0.113.10:13782
203.0.113.50:5657 in via em0
```

When `sysctl net.inet.ip.fw.verbose=0`, there is no discernible output on **ipfw0** for a **reset** action. A TCP SYN packet arrives and that is all that is displayed. To actually witness the **reset**, run [tcpdump\(8\)](#) on the specific interface (em0 in this case):

```
# tcpdump -i em0 -X -vvv
tcpdump: listening on em0, link-type EN10MB (Ethernet), capture size 262144 bytes
21:53:39.376825 IP (tos 0x0, ttl 64, id 0, offset 0, flags [DF], proto TCP (6), length
60)
    203.0.113.10.32945 > 203.0.113.50.5657: Flags [S], cksum 0x68fa (correct), seq
1926269947, win 65535, options [mss 1460,nop,wscale 6,sackOK,TS val 648984165 ecr 0],
length 0
    0x0000: 4500 003c 0000 4000 4006 ce5f ac10 0a0a  E..<..@.@.._....
    0x0010: ac10 0a32 80b1 1619 72d0 8bfb 0000 0000  ...2....r.....
    0x0020: a002 ffff 68fa 0000 0204 05b4 0103 0306  ....h.....
    0x0030: 0402 080a 26ae b665 0000 0000  ....&..e....
21:53:39.377143 IP (tos 0x10, ttl 64, id 1965, offset 0, flags [none], proto TCP (6),
length 40)
    203.0.113.50.5657 > 203.0.113.10.32945: Flags [R.], cksum 0xaddc (correct), seq 0,
ack 1926269948, win 0, length 0
    0x0000: 4510 0028 07ad 0000 4006 06b7 ac10 0a32  E..(....@.....2
```

```
0x0010:  ac10 0a0a 1619 80b1 0000 0000 72d0 8bfc  .....f...
0x0020:  5014 0000 addc 0000                      P.....
^C
```

A UDP rule containing the **reset** keyword just drops the packet. Nothing is sent back to the source address. If the **log** keyword is also used on the rule, a log entry is generated for syslog (if enabled):

```
Apr  9 21:58:49 firewall kernel: ipfw: 3000 Reset UDP 203.0.113.10:56503
203.0.113.50:5658 in via em0
```

3.3.9. Tee

The **tee** rule requires a [divert\(4\)](#) socket set up beforehand. Refer to the [divert](#) rule covered below for setting up the socket. Once the socket is set up, the **tee** keyword works like **divert** except that it is not interested in any packet return. It is simply copying the packet to the socket. Processing continues with the next rule.

In essence, **tee** allows the packet to be sent to userspace for any purpose desired - monitoring, copying, counting - whatever.

```
# ipfw add 1000 tee 700 ip from any to me
```

3.3.10. Unreach

The **unreach** keyword directs **ipfw** to respond back to the source when packets arrive with a destination port that is not opened by any service. **ipfw** sends an ICMP reply with the code set to the keyword parameter. This works for any IP protocol.

Because **ipfw** sends an ICMP packet back to the source, the ruleset must allow outbound ICMP.

Consider the following ruleset:

```
# ipfw -a list
00100 0 0 allow icmp from me to any
01000 0 0 unreach 100 log udp from any to me 5656
02000 0 0 unreach 200 log tcp from any to me 5657
03000 0 0 unreach 250 log ip from any to me 5658
```

The counters are zero when the **external1** VM sends its packet, a UDP packet destined for port 5656, for which no service is currently set up.

ipfw matches this with rule 1000 and sends an ICMP unreachable notice with code 100 (an arbitrary value, but see the list in [ipfw\(8\)](#)). The offending packet is encapsulated in the data portion of the ICMP reply:


```

# tcpdump -i bridge0 -X -vvv
tcpdump: listening on bridge0, link-type EN10MB (Ethernet), capture size 262144 bytes
10:02:37.195380 IP (tos 0x0, ttl 64, id 45085, offset 0, flags [none], proto UDP (17),
length 70)
  203.0.113.10.65216 > 203.0.113.50.5656: [udp sum ok] UDP, length 42
    0x0000:  4500 0046 b01d 0000 4011 524c cb00 710a  E..F....@.RL..q.
    0x0010:  cb00 7132 fec0 1618 0032 701e 5544 5020  ..q2.....2p.UDP.
    0x0020:  7061 636b 6574 2066 726f 6d20 5b32 3033  packet.from.[203
    0x0030:  2e30 2e31 3133 2e31 305d 2c5b 3536 3536  .0.113.10],[5656
    0x0040:  5d2c 5b32 5d0a                                ],[2].
10:02:37.196128 IP (tos 0x0, ttl 64, id 29611, offset 0, flags [none], proto ICMP (1),
length 98)
  203.0.113.50 > 203.0.113.10: ICMP 203.0.113.50 unreachable - #100, length 78
    IP (tos 0x0, ttl 64, id 45085, offset 0, flags [none], proto UDP (17), length
70)
  203.0.113.10.65216 > 203.0.113.50.5656: [udp sum ok] UDP, length 42
    0x0000:  4500 0062 73ab 0000 4001 8eb2 cb00 7132  E..bs...@.....q2
    0x0010:  cb00 710a 0364 751d 0000 0000 4500 0046  ..q..du.....E..F
    0x0020:  b01d 0000 4011 524c cb00 710a cb00 7132  ....@.RL..q...q2
    0x0030:  fec0 1618 0032 701e 5544 5020 7061 636b  .....2p.UDP.pack
    0x0040:  6574 2066 726f 6d20 5b32 3033 2e30 2e31  et.from.[203.0.1
    0x0050:  3133 2e31 305d 2c5b 3536 3536 5d2c 5b32  13.10],[5656],[2
    0x0060:  5d0a                                           ].

```

The results for a TCP unreachable are almost the same. The ICMP packet encapsulates the SYN packet in the data portion of the reply.

Here is a view of an ICMP Reply to unreachable TCP port 5657:

```

# tcpdump -i bridge0 -X -vvv
tcpdump: listening on bridge0, link-type EN10MB (Ethernet), capture size 262144 bytes
15:24:03.663104 IP (tos 0x0, ttl 64, id 0, offset 0, flags [DF], proto TCP (6), length
60)
  203.0.113.10.58575 > 203.0.113.50.5657: Flags [S], cksum 0x092c (correct), seq
1062429515, win 65535, options [mss 1460,nop,wscale 6,sackOK,TS val 3566166113 ecr 0],
length 0
    0x0000:  4500 003c 0000 4000 4006 ce5f ac10 0a0a  E..<..@.-.....
    0x0010:  ac10 0a32 e4cf 1619 3f53 634b 0000 0000  ...2....?ScK....
    0x0020:  a002 ffff 092c 0000 0204 05b4 0103 0306  ....,.....
    0x0030:  0402 080a d48f 6061 0000 0000          .....`a....
15:24:03.664168 IP (tos 0x0, ttl 64, id 37717, offset 0, flags [none], proto ICMP (1),
length 88)
  203.0.113.50 > 203.0.113.10: ICMP # 200 203.0.113.50 unreachable, length 68*
    IP (tos 0x0, ttl 64, id 0, offset 0, flags [DF], proto TCP (6), length 60)
  203.0.113.10.58575 > 203.0.113.50.5657: Flags [S], cksum 0x092c (correct), seq
1062429515, win 65535, options [mss 1460,nop,wscale 6,sackOK,TS val 3566166113 ecr 0],
length 0
    0x0000:  4500 0058 9355 0000 4001 7af3 ac10 0a32  E..X.U..@.z....2
    0x0010:  ac10 0a0a 03c8 68c3 0000 0000 4500 003c  .....h.....E..<

```

```

0x0020: 0000 4000 4006 ce5f ac10 0a0a ac10 0a32  ..@.@_.....2
0x0030: e4cf 1619 3f53 634b 0000 0000 a002 ffff  ....?ScK.....
0x0040: 092c 0000 0204 05b4 0103 0306 0402 080a  ,.....
0x0050: d48f 6061 0000 0000  ..`a....

```

3.3.11. Setdscp

The **setdscp** action directs **ipfw** to set an IP header option on outbound packets. The action has no effect on inbound packets. The header option, formerly known as the "Type of Service" (ToS) option, now defines several classes of differentiated services (DiffServ) per several RFCs - [RFC 2474](#), [RFC 3168](#), and [RFC 3260](#).

These service classes such as "Network Control", "Telephony", "Multimedia Conferencing", "Broadcast Video", "Low-latency Data", etc. all require their packets to receive special handling in the network. This is achieved by inserting "code points" - numerical values in the packet header - that define each class.

Firewalls, routers, switches, and other network devices interpret these values and, in theory, service the packets according to their class. See this Wikipedia article on Differentiated Services: https://en.wikipedia.org/wiki/Differentiated_services.

In practice, support for service classes vary among network operators. Check the man page for a list of code points settable by **ipfw**.

The example below sets the DSCP value to "af31", a codepoint in the "Multimedia Streaming" class.

On the **external1** VM, set up a listening service:

```
# ncat -u -l 5656
```

On the **firewall** VM create this ruleset:

```
# ipfw add 2000 setdscp af31 udp from me to any 5656
```

```
# ipfw add 3000 allow udp from me to any
```

Note that rule processing for the **setdscp** keyword continues to the next rule.

Note also that the DSCP value takes up only a partial byte in the IP header, sharing it with two bits of ECN (Explicit Congestion Notification). The binary value for "af31" is 011010nn, where 'nn' are the two bits for ECN. If no ECN, the value resolves to 0x68 (104 decimal).

An outbound traffic example, generated by **ncat -u 203.0.113.10 5656** from the **firewall** VM is shown as received by the **external1** VM:

On the **firewall** VM send out a UDP packet:

```
# echo "Greetings from the firewall" | ncat -u 203.0.113.10 5656
```

To view, run **tcpdump** on the network interface on the **external1** VM:

```
# tcpdump -i em0 -X -vvv
tcpdump: listening on em0, link-type EN10MB (Ethernet), capture size 262144 bytes
10:14:22.173252 IP (tos 0x68, ttl 64, id 30816, offset 0, flags [none], proto UDP
(17), length 57)
  203.0.113.50.17767 > 203.0.113.10.5656: [udp sum ok] UDP, length 29
    0x0000: 4568 0039 7860 0000 4011 958f ac10 0a32  Eh.9x`..@.....2
    0x0010: ac10 0a0a 4567 1618 0025 0f5b 4772 6565  ....Eg...%.[Gree
    0x0020: 7469 6e67 7320 6672 6f6d 2074 6865 2066  tings.from.the.f
    0x0030: 6972 6577 616c 6c2e 0a                irewall..
```

Diffserve codepoints can be set on any IP based protocol or restricted to selected protocols and/or ports through suitable rules.

3.3.12. Skipto

The **skipto** action directs the firewall engine to pass over any rules less than the **skipto** parameter number. If an early rule can match a packet characteristic such as an address, port, TCP or UDP header option or similar, a **skipto** rule can jump to a potentially much later section of the firewall ruleset to handle the packet.

Consider the following (contrived) ruleset:

```
# ipfw add 100 check-state
# ipfw add 1000 allow tcp from me to any established keep-state
# ipfw add 2000 allow tcp from 203.0.113.10 to me 4500 setup keep-state
# ipfw add 3000 allow tcp from 203.0.113.10 to me 4502 setup keep-state
# ipfw add 4000 allow tcp from 203.0.113.10 to me 4504 setup keep-state
# ipfw add 5000 allow tcp from 203.0.113.10 to me 4506 setup keep-state
# ipfw add 6000 allow tcp from 203.0.113.10 to me 4508 setup keep-state
# ipfw add 7000 allow tcp from 203.0.113.10 to me 4510 setup keep-state
# ipfw add 8000 allow tcp from 203.0.113.10 to me 4512 setup keep-state
# ipfw add 9000 allow tcp from 203.0.113.10 to me 4512 setup keep-state
# ipfw add 10000 allow tcp from 203.0.113.10 to me 5656 setup keep-state
```

With the **external1** VM using the **tcon.sh 5656** TCP connection script, **ipfw** has to traverse the entire firewall ruleset, checking each rule in turn for a match. (When testing this ruleset, ensure that the **firewall** VM is running the appropriate service script such as **tserv3.sh**.)

By placing a **skipto** action rule after the **check-state** action, **ipfw** jumps directly to the desired rule:

```
# ipfw add 100 check-state
# ipfw add 500 skipto 10000 tcp from 203.0.113.10 to me 5656
# ipfw add 1000 allow tcp from me to any established keep-state
# ipfw add 2000 allow tcp from 203.0.113.10 to me 4500 setup keep-state
# ipfw add 3000 allow tcp from 203.0.113.10 to me 4502 setup keep-state
# ipfw add 4000 allow tcp from 203.0.113.10 to me 4504 setup keep-state
# ipfw add 5000 allow tcp from 203.0.113.10 to me 4506 setup keep-state
# ipfw add 6000 allow tcp from 203.0.113.10 to me 4508 setup keep-state
```

```
# ipfw add 7000 allow tcp from 203.0.113.10 to me 4510 setup keep-state
# ipfw add 8000 allow tcp from 203.0.113.10 to me 4512 setup keep-state
# ipfw add 9000 allow tcp from 203.0.113.10 to me 4512 setup keep-state
# ipfw add 10000 allow tcp from 203.0.113.10 to me 5656 setup keep-state
```

Use the `-a` command line parameter to see if the **skipto** action is working (or use **ipfw show**):

```
# ipfw -a list
00100 0 0 check-state :default
00500 1 60 skipto 10000 tcp from 203.0.113.10 to me 5656
01000 0 0 allow tcp from me to any established keep-state :default
02000 0 0 allow tcp from 203.0.113.10 to me 4500 setup keep-state :default
03000 0 0 allow tcp from 203.0.113.10 to me 4502 setup keep-state :default
04000 0 0 allow tcp from 203.0.113.10 to me 4504 setup keep-state :default
05000 0 0 allow tcp from 203.0.113.10 to me 4506 setup keep-state :default
06000 0 0 allow tcp from 203.0.113.10 to me 4508 setup keep-state :default
07000 0 0 allow tcp from 203.0.113.10 to me 4510 setup keep-state :default
08000 0 0 allow tcp from 203.0.113.10 to me 4512 setup keep-state :default
09000 0 0 allow tcp from 203.0.113.10 to me 4514 setup keep-state :default
10000 8 474 allow tcp from 203.0.113.10 to me 5656 setup keep-state :default
65535 0 0 deny ip from any to any
```

General notes on skipto:

- The **skipto** action does not allow negative numbers as a parameter.
- A **skipto** to rule 0 or to a value greater than 65534, causes **ipfw** to throw an error.
- It is possible to use the **skipto** action to skip between **sets**. However, if the **set** containing the **skipto** target is disabled, processing continues with the next rule in **any set** that is enabled.

For example, if there are three **sets** - 0, 1, and 2, with a disabled **set** 1 containing the destination of the **skipto** action, processing will continue with the next rule. See the below ruleset and counters. Because **set** 1 is disabled, the next rule in any enabled **set** is rule 2700. Processing continues at 2700, but the packet was not matched until rule 3000.

```
# ipfw -Sa list
00100 0 0 set 0 check-state :default
00101 1 60 set 0 skipto 2000 tcp from 203.0.113.10 to me
00120 0 0 set 0 allow tcp from 203.0.113.10 to me 5656 setup keep-state :default
00150 0 0 set 0 allow tcp from me to any established keep-state :default
01000 0 0 set 0 allow tcp from 203.0.113.10 to me 5656 setup keep-state :default
# DISABLED 01200 0 0 set 1 allow tcp from 203.0.113.10 to me 6500 setup keep-state
:default
# DISABLED 01800 0 0 set 1 allow tcp from 203.0.113.10 to me 6512 setup keep-state
:default
# DISABLED 01900 0 0 set 1 allow tcp from 203.0.113.10 to me 6514 setup keep-state
:default
# DISABLED 02000 0 0 set 1 allow tcp from 203.0.113.10 to me 5656 setup keep-state
:default
```

```

02700 0 0 set 2 allow tcp from 203.0.113.10 to me 7510 setup keep-state :default
02800 0 0 set 2 allow tcp from 203.0.113.10 to me 7512 setup keep-state :default
02900 0 0 set 2 allow tcp from 203.0.113.10 to me 7514 setup keep-state :default
03000 8 474 set 2 allow tcp from 203.0.113.10 to me 5656 setup keep-state :default
65535 0 0 set 31 deny ip from any to any

```

- If using **skipto** to a rule number that has multiple rules, the first matching rule at or after that number is executed:

```

# ipfw -Sa list
00100 0 0 set 0 check-state :default
00101 1 60 set 0 skipto 2000 tcp from 203.0.113.10 to me
00120 0 0 set 0 allow tcp from 203.0.113.10 to me 5656 setup keep-state :default
00150 0 0 set 0 allow tcp from me to any established keep-state :default
01600 0 0 set 1 allow tcp from 203.0.113.10 to me 6508 setup keep-state :default
01700 0 0 set 1 allow tcp from 203.0.113.10 to me 6510 setup keep-state :default
02000 0 0 set 1 allow tcp from 203.0.113.10 to me 6512 setup keep-state :default
02000 0 0 set 1 allow tcp from 203.0.113.10 to me 6514 setup keep-state :default
02000 0 0 set 2 allow tcp from 203.0.113.10 to me 7500 setup keep-state :default
02000 0 0 set 2 allow tcp from 203.0.113.10 to me 7502 setup keep-state :default
02000 0 0 set 2 allow tcp from 203.0.113.10 to me 7504 setup keep-state :default
02500 0 0 set 2 allow tcp from 203.0.113.10 to me 7506 setup keep-state :default
02600 0 0 set 2 allow tcp from 203.0.113.10 to me 7508 setup keep-state :default
02700 0 0 set 2 allow tcp from 203.0.113.10 to me 7510 setup keep-state :default
02800 0 0 set 2 allow tcp from 203.0.113.10 to me 7512 setup keep-state :default
02900 0 0 set 2 allow tcp from 203.0.113.10 to me 7514 setup keep-state :default
03000 10 567 set 2 allow tcp from 203.0.113.10 to me 5656 setup keep-state :default
65535 0 0 set 31 deny ip from any to any

```

- It is possible to enter a rule with a **skipto** rule number that is lower than the current rule number, attempting to go backward in the ruleset. However, this has no effect, and processing continues with the next rule:

```

# ipfw -Sa list
00100 0 0 set 0 check-state :default
00101 1 60 set 0 skipto 1000 tcp from 203.0.113.10 to me
00120 0 0 set 0 allow tcp from 203.0.113.10 to me 5656 setup keep-state :default
00150 0 0 set 0 allow tcp from me to any established keep-state :default
00200 0 0 set 0 allow tcp from 203.0.113.10 to me 4500 setup keep-state :default
00300 0 0 set 0 allow tcp from 203.0.113.10 to me 4502 setup keep-state :default
00400 0 0 set 0 allow tcp from 203.0.113.10 to me 4504 setup keep-state :default
00500 0 0 set 0 allow tcp from 203.0.113.10 to me 5656 setup keep-state :default
00600 0 0 set 0 allow tcp from 203.0.113.10 to me 4508 setup keep-state :default
00700 0 0 set 0 allow tcp from 203.0.113.10 to me 4512 setup keep-state :default
00800 0 0 set 0 allow tcp from 203.0.113.10 to me 4514 setup keep-state :default
01000 1 60 set 0 skipto 500 tcp from 203.0.113.10 to me
01100 8 475 set 0 allow tcp from 203.0.113.10 to me 5656 setup keep-state :default
65535 0 0 set 31 deny ip from any to any

```

- It is also possible (but not advised) to **skipto** a **skipto** rule:

```
00100 0 0 set 0 check-state :default
00101 1 60 set 0 skipto 1000 tcp from 203.0.113.10 to me
00120 0 0 set 0 allow tcp from 203.0.113.10 to me 5656 setup keep-state :default
00150 0 0 set 0 allow tcp from me to any established keep-state :default
00200 0 0 set 0 allow tcp from 203.0.113.10 to me 4500 setup keep-state :default
00300 0 0 set 0 allow tcp from 203.0.113.10 to me 4502 setup keep-state :default
00400 0 0 set 0 allow tcp from 203.0.113.10 to me 4504 setup keep-state :default
00500 0 0 set 0 allow tcp from 203.0.113.10 to me 5656 setup keep-state :default
00600 0 0 set 0 allow tcp from 203.0.113.10 to me 4508 setup keep-state :default
00700 0 0 set 0 allow tcp from 203.0.113.10 to me 4510 setup keep-state :default
00800 0 0 set 0 allow tcp from 203.0.113.10 to me 4512 setup keep-state :default
00900 0 0 set 0 allow tcp from 203.0.113.10 to me 4514 setup keep-state :default
01000 1 60 set 0 skipto 1500 tcp from 203.0.113.10 to me
01000 0 0 set 0 allow tcp from 203.0.113.10 to me 5656 setup keep-state :default
01500 1 60 set 0 skipto 2000 tcp from 203.0.113.10 to me
01600 0 0 set 0 allow tcp from 203.0.113.10 to me 5656 setup keep-state :default
02000 1 60 set 0 skipto 2500 tcp from 203.0.113.10 to me
02100 0 0 set 0 allow tcp from 203.0.113.10 to me 5656 setup keep-state :default
02500 1 60 set 0 skipto 3000 tcp from 203.0.113.10 to me
02600 0 0 set 0 allow tcp from 203.0.113.10 to me 5656 setup keep-state :default
03000 8 475 set 0 allow tcp from 203.0.113.10 to me 5656 setup keep-state :default
65535 0 0 set 31 deny ip from any to any
```

See the section on Lists for additional caveats.

3.3.13. Divert

The **divert** mechanism in **ipfw** allows **ipfw** to pull packets into user space for programmatic purposes. The **divert** rule snatches the packet and presents it to a **divert(4)** socket, a special socket type that can be created from an external program. See the *divert.c* program at the end of [Appendix B](#) for the sample program used for this book. Copy the **divert.c** program onto the **firewall** VM from the host and compile it:

```
Copy divert.c from the host:
# cd /root/bin
# scp user@host:~/ipfw-primer/ipfw/VM_CODE/divert.c .

# Compile it:
# make divert LDFLAGS=-lutil
```

The code should compile cleanly. If it does not, examine the file closely to ensure it was copied correctly and retry the above command.

Divert sockets can be used as the basis for many specialized applications such as packet examination, in-flight packet modification, experimental routing techniques, etc. The program shown here simply reads from the socket and dumps the contents of the packet in hex and ASCII. It

then writes the packet back into the divert socket.

To work with the **divert** keyword, the [divert\(4\)](#) packet diversion mechanism has to be compiled into the kernel or loaded at runtime:

```
# kldload ipfw
# kldload ipdivert
```

This loads the **ipfw** firewall kernel module and the **ipdivert** kernel module which provides [divert\(4\)](#) functionality.



Once loaded, the **ipdivert.ko** module cannot be unloaded. A firewall reboot is required to remove the **ipdivert.ko** module.

Once this is done, an application can open a [divert\(4\)](#) socket and process packets.

```
# ./divert
Opening divert on port 700

... (see below)
```

In another window, run [netstat\(1\)](#) to see the divert socket:

```
# netstat -an | more
Active Internet connections (including servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         (state)
tcp4      0      0 *.22                    *.*                     LISTEN
tcp6      0      0 *.22                    *.*                     LISTEN
udp4      0      0 *.514                   *.*                     
udp6      0      0 *.514                   *.*                     
div4      0      0 *.700                   *.*                     
Active UNIX domain sockets
Address          Type  Recv-Q Send-Q           Inode          Conn
Refs           Nextref Addr
fffff80003b83000 stream  0      0 fffff80003cac5a0 0
0               0 /var/run/devd.pipe
fffff80003baf800 dgram  0      0               0 fffff80003bafc00
0 fff
...
```

To examine the **divert** operation, first create a suitable ruleset:

```
# ipfw add 700 divert 700 ip from any to any
#
# ipfw add 1000 allow udp from 203.0.113.10 to me
01000 allow udp from 203.0.113.10 to me
```

```
#
# ipfw add 1100 allow udp from me to 203.0.113.10
01100 allow udp from me to 203.0.113.10
```

The syntax is a bit odd in this case. The `divert` keyword takes a numeric argument that functions as the divert object. This is similar syntax to **pipes**, **queues**, and **NAT** (network address translation) rules which are discussed later.

A common convention, though not required, is to make the **divert** port the same number as the rule number in the ruleset. Whatever the rule number, when the packet is diverted and processed, and then returned to **ipfw**, the firewall picks up the packet at the divert rule number, plus one - that is, the next rule.

```
# ipfw list
00700 divert 700 ip from any to any
01000 allow udp from 203.0.113.10 to me
01100 allow udp from me to 203.0.113.10
65535 deny ip from any to any
```

Examine the ruleset above. The packet is diverted to a divert socket, port 700 at the first rule. When it is returned from the `divert.c` program, it reenters the ruleset at rule 1000. The ruleset allows UDP packets from and to the **external1** VM.

To test, set up the **firewall** VM host to run **userv3.sh** and the **external1** VM host to run **ucon.sh 5656**. This results in the following expected output from the divert program:

```
# ./divert
Opening divert on port 700
203.0.113.10:51417 -> 203.0.113.50:5656
|0000 45 00 00 46 a9 0c 00 00 40 11 65 3e ac 10 0a 0a |E..F....@.e>....|
|0010 ac 10 0a 32 c8 d9 16 18 00 32 a2 eb 55 44 50 20 |...2.....2..UDP |
|0020 70 61 63 6b 65 74 20 66 72 6f 6d 20 5b 31 37 32 |packet from [172|
|0030 2e 31 36 2e 31 30 2e 31 30 5d 2c 5b 35 36 35 36 |.16.10.10],[5656|
|0040 5d 2c 5b 34 5d 0a |],[4]. |
```

And the output from **userv3.sh** also shows on the **firewall** console:

```
# sh userv3.sh
Starting UDP listeners on [5656],[5657],[5658]
UDP communication from [203.0.113.10],[5656],[1]
```

Next, shut down the **userv3.sh** services on the **firewall** VM. The incoming packets find no open port and are rejected by the **firewall** VM host. However, they still go through the divert socket:

```
# ./divert
Opening divert on port 700
```



```

203.0.113.10:26058 -> 203.0.113.50:5656
|0000 45 00 00 47 a9 12 00 00 40 11 65 37 ac 10 0a 0a |E..G....@.e7....|
|0010 ac 10 0a 32 65 ca 16 18 00 33 28 a9 55 44 50 20 |...2e....3(.UDP |
|0020 70 61 63 6b 65 74 20 66 72 6f 6d 20 5b 31 37 32 |packet from [172|
|0030 2e 31 36 2e 31 30 2e 31 30 5d 2c 5b 35 36 35 36 |.16.10.10],[5656|
|0040 5d 2c 5b 31 30 5d 0a |],[10]. |
203.0.113.50:771 -> 203.0.113.10:27038
|0000 45 00 00 63 3a 61 00 00 40 01 65 37 ac 10 0a 32 |E..c:a..@.e7...2|
|0010 ac 10 0a 0a 03 03 69 9e 00 00 00 00 45 00 00 47 |.....i.....E..G|
|0020 a9 12 00 00 40 11 65 37 ac 10 0a 0a ac 10 0a 32 |....@.e7.....2|
|0030 65 ca 16 18 00 33 28 a9 55 44 50 20 70 61 63 6b |e....3(.UDP pack|
|0040 65 74 20 66 72 6f 6d 20 5b 31 37 32 2e 31 36 2e |et from [172.16.|
|0050 31 30 2e 31 30 5d 2c 5b 35 36 35 36 5d 2c 5b 31 |10.10],[5656],[1|
|0060 30 5d 0a |0]. |
divert: sendto: Permission denied

```

The last output line, "Permission denied", is because the kernel, faced with a packet and no port to send it to, instead sends an ICMP port unreachable response back to the sender. The kernel tries to send the ICMP packet back out the network interface, but there is no **ipfw** rule for it through the firewall - thus "Permission denied". The packet is dropped.

To fix, add a rule for ICMP traffic in either direction:

```

# ipfw list
00700 divert 700 ip from any to any
00800 allow icmp from any to any
01000 allow udp from 203.0.113.10 to me
01100 allow udp from me to 203.0.113.10
65535 deny ip from any to any

```

The **divert** operation now works as expected and the packet re-enters the firewall after rule 700. The next rule (800) permits ICMP in either direction and the packet is sent back to the source host. In the listings below, the **ucon.sh** script was run for 5 cycles, and after cycle #3, the firewall **userv3.sh** script was shut down.

The remaining two cycles result in an ICMP message being returned back to the **external1** VM:

```

# ./divert
Opening divert on port 700
203.0.113.10:36083 -> 203.0.113.50:5656
|0000 45 00 00 46 a9 20 00 00 40 11 65 2a ac 10 0a 0a |E..F. ..@.e*....|
|0010 ac 10 0a 32 8c f3 16 18 00 32 de d4 55 44 50 20 |...2....2..UDP |
|0020 70 61 63 6b 65 74 20 66 72 6f 6d 20 5b 31 37 32 |packet from [172|
|0030 2e 31 36 2e 31 30 2e 31 30 5d 2c 5b 35 36 35 36 |.16.10.10],[5656|
|0040 5d 2c 5b 31 5d 0a |],[1]. |
203.0.113.10:25662 -> 203.0.113.50:5656
|0000 45 00 00 46 a9 21 00 00 40 11 65 29 ac 10 0a 0a |E..F.!...@.e)....|
|0010 ac 10 0a 32 64 3e 16 18 00 32 07 89 55 44 50 20 |...2d>...2..UDP |

```

```

|0020  70 61 63 6b 65 74 20 66 72 6f 6d 20 5b 31 37 32 |packet from [172|
|0030  2e 31 36 2e 31 30 2e 31 30 5d 2c 5b 35 36 35 36 |.16.10.10],[5656|
|0040  5d 2c 5b 32 5d 0a                                     |],[2].           |
203.0.113.10:40345 -> 203.0.113.50:5656
|0000  45 00 00 46 a9 22 00 00 40 11 65 28 ac 10 0a 0a |E..F."..@.e(....|
|0010  ac 10 0a 32 9d 99 16 18 00 32 ce 2c 55 44 50 20 |...2.....2.,UDP |
|0020  70 61 63 6b 65 74 20 66 72 6f 6d 20 5b 31 37 32 |packet from [172|
|0030  2e 31 36 2e 31 30 2e 31 30 5d 2c 5b 35 36 35 36 |.16.10.10],[5656|
|0040  5d 2c 5b 33 5d 0a                                     |],[3].           |
203.0.113.10:53482 -> 203.0.113.50:5656
|0000  45 00 00 46 a9 23 00 00 40 11 65 27 ac 10 0a 0a |E..F.x..@.e'....|
|0010  ac 10 0a 32 d0 ea 16 18 00 32 9a da 55 44 50 20 |...2.....2..UDP |
|0020  70 61 63 6b 65 74 20 66 72 6f 6d 20 5b 31 37 32 |packet from [172|
|0030  2e 31 36 2e 31 30 2e 31 30 5d 2c 5b 35 36 35 36 |.16.10.10],[5656|
|0040  5d 2c 5b 34 5d 0a                                     |],[4].           |
203.0.113.50:771 -> 203.0.113.10:27037 (ICMP packet)
|0000  45 00 00 62 3a 68 00 00 40 01 65 27 ac 10 0a 32 |E..b:h..@.e'...2|
|0010  ac 10 0a 0a 03 03 69 9d 00 00 00 00 45 00 00 46 |.....i.....E..F|
|0020  a9 23 00 00 40 11 65 27 ac 10 0a 0a ac 10 0a 32 |.x..@.e'.....2|
|0030  d0 ea 16 18 00 32 9a da 55 44 50 20 70 61 63 6b |.....2..UDP pack|
|0040  65 74 20 66 72 6f 6d 20 5b 31 37 32 2e 31 36 2e |et from [172.16.|
|0050  31 30 2e 31 30 5d 2c 5b 35 36 35 36 5d 2c 5b 34 |10.10],[5656],[4|
|0060  5d 0a                                               |].               |
203.0.113.10:35359 -> 203.0.113.50:5656
|0000  45 00 00 46 a9 24 00 00 40 11 65 26 ac 10 0a 0a |E..F.$..@.e&....|
|0010  ac 10 0a 32 8a 1f 16 18 00 32 e1 a4 55 44 50 20 |...2.....2..UDP |
|0020  70 61 63 6b 65 74 20 66 72 6f 6d 20 5b 31 37 32 |packet from [172|
|0030  2e 31 36 2e 31 30 2e 31 30 5d 2c 5b 35 36 35 36 |.16.10.10],[5656|
|0040  5d 2c 5b 35 5d 0a                                     |],[5].           |
203.0.113.50:771 -> 203.0.113.10:27037 (ICMP packet)
|0000  45 00 00 62 3a 69 00 00 40 01 65 26 ac 10 0a 32 |E..b:i..@.e&...2|
|0010  ac 10 0a 0a 03 03 69 9d 00 00 00 00 45 00 00 46 |.....i.....E..F|
|0020  a9 24 00 00 40 11 65 26 ac 10 0a 0a ac 10 0a 32 |.$.@.e&.....2|
|0030  8a 1f 16 18 00 32 e1 a4 55 44 50 20 70 61 63 6b |.....2..UDP pack|
|0040  65 74 20 66 72 6f 6d 20 5b 31 37 32 2e 31 36 2e |et from [172.16.|
|0050  31 30 2e 31 30 5d 2c 5b 35 36 35 36 5d 2c 5b 35 |10.10],[5656],[5|
|0060  5d 0a                                               |].               |
^C

```

Note the two icmp packets logged by rule 800:

```

# ipfw show
00700 19 2466 divert 700 ip from any to any
00800 2 196 allow icmp from any to any
01000 5 350 allow udp from 203.0.113.10 to me
01100 0 0 allow udp from me to 203.0.113.10
65535 477 114991 deny ip from any to any

```

General notes on the divert action:

- The **ipdivert.ko** kernel module must be loaded or compiled into the kernel to create a divert rule, and thus to use `divert(4)` sockets.
- The **ipdivert.ko** kernel module cannot be unloaded. Restart the VM to remove the **ipdivert.ko** kernel module.
- It is not possible to create a rule with a divert port of 0 or 65535. The port number must be between 1 and 65534 (inclusive).
- If creating a rule with a divert port on rule 65534, the returning packet will restart firewall rule processing at the default rule, 65535, which cannot be changed.
- A divert rule can be created for any protocol in `/etc/protocols`.
- The same divert port can be used for multiple rules.
- After returning from a divert rule, if the next rule is in another set, processing will continue with that rule unless the set is disabled. If disabled, it will skip to the next rule in any set that is not disabled.

General notes on creating the divert socket:

- Only root can create a divert socket.
- Opening a divert socket on port 0 or port 65536 results in a random divert port number.
- Opening a divert socket on port 65535 is permitted, but not advised.
- Opening a divert port greater than 65536 or less than 0 results in a positive port number modulo 65536.
- As with other sockets, it is not possible to open two divert sockets on the same port number. However, it is possible to open a divert socket on a port already in use for any protocols based on IPv4 or IPv6.

3.3.14. Other Protocols

Any protocol in `/etc/protocols` may be used in a rule.

```
# ipfw add 1000 allow ospf from any to me
# ipfw add 2000 allow chaos from any to me
etc.
```

3.3.15. Limit

ipfw can restrict the number of active connections with the **limit** option. This option allows for specifying a parameter in the rule that is regarded as a flow element, that is, one of `src-addr`, `src-port`, `dst-addr`, or `dst-port`. In addition, the **limit** keyword takes a value, N, that is the maximum number of connections desired:

```
# ipfw add 1000 allow udp from any to me dst-port 5656-5658 limit src-addr 5
# ipfw add 1100 count udp from any to me
```

Concurrent connections via TCP, UDP, ICMP, or any protocol can be limited in this way.

ipfw creates a dynamic rule for each connection allowed by the rule. When the maximum number of connections is reached, additional packets are considered no longer matched and are dropped by the rule after being counted, and the search terminates.

To test, write the following simple script on the **external1** VM to flood UDP packets at the **firewall** VM running **sh userv3.sh**:

```
#!/bin/sh

export NUM=1
for i in `jot -r 500 5656 5658 1`
do
    echo "hello [${NUM}] to port [${i}]" | ncat -u 203.0.113.50 ${i}
    NUM=expr $NUM + 1
done
```

On the **firewall** VM, **ipfw** starts creating dynamic rules as soon as the first matching packet is received. Additional dynamic rules, up to the **limit** number are created. Each UDP based dynamic rule has a default 10 second lifetime, controlled by the sysctl node `net.inet.ip.fw.dyn_udp_lifetime`. As they expire under the limit value, space for additional connections is created. The number of open dynamic rules at any point in time can be viewed with the sysctl node `net.inet.ip.fw.dyn_count`.

View the dynamic rules with:

```
# ipfw -SaD list
00500 0 0 check-state :default
01000 10 531 allow udp from any to me 5656-5658 limit src-addr 5 :default
65535 0 0 deny ip from any to any
## Dynamic rules (4 560):
01000 1 53 (8s) LIMIT udp 203.0.113.10 23755 <-> 203.0.113.50 5657 :default
01000 1 53 (8s) LIMIT udp 203.0.113.10 30144 <-> 203.0.113.50 5656 :default
01000 0 0 (4s) PARENT 3 udp 203.0.113.10 0 <-> 0.0.0.0 0 :default
01000 1 53 (8s) LIMIT udp 203.0.113.10 22722 <-> 203.0.113.50 5658 :default
```



Running `cmdwatch -n1 ipfw -SaD list` on the **firewall** VM will show the list of rules grow and shrink in real time.

It is useful to experiment with the sysctl `net.inet.ip.fw.dyn_udp_lifetime` and see its effect on `net.inet.ip.fw.dyn_count`. By adjusting the `net.inet.ip.fw.dyn_udp_lifetime` value during a network packet flood (like that above), it is possible to watch how the **ipfw limit** rule blocked traffic through the firewall.

Here is the result of a sample run. Note the missing connections due to a limit restriction:

```
...
hello [19] to port [5657]
hello [20] to port [5656]
hello [21] to port [5656]
hello [33] to port [5657]
hello [34] to port [5656]
hello [35] to port [5657]
hello [36] to port [5657]
hello [37] to port [5657]
hello [48] to port [5657]
hello [50] to port [5657]
hello [51] to port [5656]
hello [52] to port [5657]
hello [53] to port [5656]
hello [64] to port [5657]
...
```

3.3.16. Call and Return

The **call** and **return** actions allow **ipfw** to change ruleset processing order by jumping to a rule number elsewhere in the ruleset. If the rules at that location contain a **return** action, processing will jump back to the statement immediately after the original **call** statement. In practice, this acts like a program function call, or as [ipfw\(8\)](#) notes, like an assembly language subroutine.

Creating a new ruleset with **call** and **return** actions:

```
#
# ipfw add 500 check-state
00500 check-state :default
#
# ipfw add 1000 call 20000 udp from 203.0.113.10 to me 5656
01000 call 20000 udp from 203.0.113.10 to me 5656
#
# ipfw add 1100 count udp from 203.0.113.10 to me
01100 count udp from 203.0.113.10 to me
#
# ipfw add 1200 allow udp from 203.0.113.10 to me 5656
01200 allow udp from 203.0.113.10 to me 5656
#
# ipfw add 20000 count udp from 203.0.113.10 to me
20000 count udp from 203.0.113.10 to me
#
# ipfw add 21000 return via any
21000 return
#
# ipfw -a list
00500 0 0 check-state :default
01000 0 0 call 20000 udp from 203.0.113.10 to me 5656
01100 0 0 count udp from 203.0.113.10 to me
```

```
01200 0 0 allow udp from 203.0.113.10 to me 5656
20000 0 0 count udp from 203.0.113.10 to me
21000 0 0 return
65535 0 0 deny ip from any to any
```

As noted in the man page, some extra syntactic sugar on the return statement is required:

```
# ipfw add 21000 return via any
```

In this example several count statements are used to try to trace ruleset processing.

To test, have the **firewall** VM host startup **user3.sh**. After the **external1** VM uses **ucon.sh 5656** to send a single udp packet, the count list looks like this:

```
# ipfw -a list
00500 0 0 check-state :default
01000 1 70 call 20000 udp from 203.0.113.10 to me 5656
01100 1 70 count udp from 203.0.113.10 to me
01200 1 70 allow udp from 203.0.113.10 to me 5656
20000 1 70 count udp from 203.0.113.10 to me
21000 1 70 return via any
65535 0 0 deny ip from any to any
```

The packet was matched at rule 1000 where it encountered a **call** action to jump to rule 20000 where it was then matched. The next rule was a matched **return** action at rule 21000. Returning to the rule after the **call** action, it matched a **count** action at 1100, then matched an **allow** action at 1200 where it was sent through to the application layer and was received by **user3.sh**.

If the **return** action at rule 21000 is removed and the test is re-run, the counts look much different. (First, **ipfw zero** and **ipfw zero 65535** to reset all counters.)

```
# ipfw -a list
00500 0 0 check-state :default
01000 1 70 call 20000 udp from 203.0.113.10 to me 5656
01100 0 0 count udp from 203.0.113.10 to me
01200 0 0 allow udp from 203.0.113.10 to me 5656
20000 1 70 count udp from 203.0.113.10 to me
65535 1 70 deny ip from any to any
```

Without a **return** at rule 21000, the only rule left is the default **deny** rule, and there is nothing received by **user3.sh**.



Because **ipfw** can jump both forward and backward with the **call** action, it is possible to create an endless loop.

This example creates an endless loop with an incorrect **call** rule:

```
#
# ipfw -a list
01000 0 0 count udp from 203.0.113.10 to me
05000 0 0 check-state :default
06000 0 0 call 1000 udp from 203.0.113.10 to me 5656
07000 0 0 count udp from 203.0.113.10 to me
65535 0 0 deny ip from any to any
```

This example is missing a **return** action. This creates a loop. **ipfw** eventually figures out that a loop exists, and breaks out at the next call action with the diagnostic:

```
# ipfw: call stack error, go to next rule
```



ipfw does its best to get your attention for this error. The above diagnostic shows up on the console, in any console log, and in the `/var/log/messages` file regardless of the state of `sysctl net.inet.ip.fw.verbose`.

Unfortunately, **ipfw** does not currently note *where* the missing **return** action is or which rule it went to next.

It is possible to pick up a clue by watching the rule counts. Below is the rule count for this errant ruleset after just one packet was received from the **external1** VM:

```
# ipfw -a list
01000 17 1207 count udp from 203.0.113.10 to me
05000 0 0 check-state :default
06000 16 1136 call 1000 udp from 203.0.113.10 to me 5656
07000 1 71 count udp from 203.0.113.10 to me
65535 1 71 deny ip from any to any
```

This shows that **ipfw** went around this **call** loop 16 times before throwing an error.

For TCP connections, call and return operate almost the same. Below is a ruleset with TCP instead of UDP for the desired protocol and including the required **setup** and **keep-state** keywords:

```
# ipfw -a list
00500 0 0 check-state :default
01000 0 0 call 20000 tcp from 203.0.113.10 to me 5656
01100 0 0 count tcp from 203.0.113.10 to me
02000 0 0 allow tcp from 203.0.113.10 to me 5656 setup keep-state :default
20000 0 0 count tcp from 203.0.113.10 to me
21000 0 0 return
65535 0 0 deny ip from any to any
```

After the connection is successfully made from external1, the observed counts are:

```
# ipfw -a list
00500 0 0 check-state :default
01000 1 60 call 20000 tcp from 203.0.113.10 to me 5656
01100 1 60 count tcp from 203.0.113.10 to me
02000 8 479 allow tcp from 203.0.113.10 to me 5656 setup keep-state :default
20000 1 60 count tcp from 203.0.113.10 to me
21000 1 60 return
65535 0 0 deny ip from any to any
```

The difference in rule counts is due to dynamic rules created by the **setup** and **keep-state** keywords on rule 2000. Counts can also be shown with the **-d** command line parameter. The numbers below are from just the initial 3-way handshake:

```
# ipfw -ad list
00500 0 0 check-state :default
01000 1 60 call 20000 tcp from 203.0.113.10 to me 5656
01100 1 60 count tcp from 203.0.113.10 to me
02000 5 276 allow tcp from 203.0.113.10 to me 5656 setup keep-state :default
20000 1 60 count tcp from 203.0.113.10 to me
21000 1 60 return
65535 0 0 deny ip from any to any
## Dynamic rules (1 152):
02000 5 276 (296s) STATE tcp 203.0.113.10 19179 <-> 203.0.113.50 5656 :default
```

The above numbers indicate that the 3-way handshake occurred during the dynamic rule setup.

General notes on call and return:

- **ipfw** allows a **call** out to an address either before or after the current rule.
- There must be a **return** for every **call**.
- **call** / **return** pairs can be nested up to 16 levels deep. If **ipfw** sees one more **call** rule, it will throw the "call stack error, go to next rule" error and continue with the next rule.
- Similar to **skip-to**, a **call** to rule 0 or to a value greater than 65534, causes **ipfw** to throw an error.
- Similar also to **sets**, if a **call** is made to a target rule in a set that is disabled, the call will land on the next rule in any non-disabled set and processing continues from there.
- If a **return** is encountered when no **call** has been made, the **return** rule is ignored and processing continues with the next rule.

3.3.17. Using uid and gid in rules

An interesting capability of **ipfw** is its ability to match packets on Unix **uid** and **gid** values. Network packets themselves have no inherent ownership, so where does this capability come from? Answer - it comes from the applications that are the source and destination of those packets.

The new ruleset below is a simple example to examine this capability. First, locate or add a user (here 'quarven') if needed.

Then, note the syntax needed for **uid** / **gid** matching:

```
# grep quarven /etc/passwd
quarven:*:1002:1002:Quarven:/home/quarven:/bin/sh

Copy userv.sh to user quarven home directory
# cp ~/bin/userv.sh /home/quarven

Now use userid 'quarven' in an ipfw rule:

# ipfw add 700 allow udp from 203.0.113.10 to me uid quarven
00700 allow udp from 203.0.113.10 to me uid quarven
#
# ipfw show
00700 0 0 allow udp from 203.0.113.10 to me uid quarven
65535 0 0 deny ip from any to any
```

Next, login as user 'quarven' and run the script `~/userv.sh 5656`. Then switch to the **external1** VM and run `echo "hello there" | ncat -u 203.0.113.50 5656`. The results should appear on the console running the **userv.sh** script.

The results are shown below.

```
quarven@firewall:~ $ /bin/sh userv.sh 5656
PORT1 = [5656]
Starting UDP listener on [203.0.113.50],[5656]
hello there
^Cquarven@firewall:~ $
```

This works because the instance of **userv.sh** is run under the **uid** quarven as shown below:

```
Show the user information for the userv.sh instance:

root@firewall:~ # ps -o user -xl -U quarven
USER      UID   PID  PPID  C  PRI  NI   VSZ  RSS  MWCHAN  STAT  TT      TIME  COMMAND
quarven  1002  4256  83703 0   61   0  13380 2908  pause  I+    v0    0:00.03  sh userv.sh 5656
quarven  1002  6878  4256  0   61   0  13400 2344  select I+    v0    0:00.02  nc -l -k -u
203.0.113.50 5656
quarven  1002  83703 83431 0   20   0  13380 3168  wait   I     v0    0:00.10  -sh (sh)
```

ipfw has matched an incoming network packet to a program owned by a userid. If the rule is changed to another userid, even **root**, the match will not succeed and the packet will be picked up by the default rule. Likewise, if the **userv.sh** script is run under another user, even **root**, the match will not succeed.



It is sometimes necessary to immediately shut down all IP traffic to or from a certain user. This capability can be used for that purpose. Note however, that the **deny** rule below must come **before any check-state rule** to catch traffic that may be otherwise allowed by a dynamic rule.

```
# ipfw add 50 deny ip from any to any uid quarven
# ipfw add 100 check-state
```

Note also the item about ICMP traffic in the "General Notes" below.

General Notes on Using uid and gid:

- The **gid** keyword works in an identical fashion to **uid** described above.
- If using a name as the **uid** or **gid**, the name must exist in the indicated system file.
- Ranges and lists of **uids** or **gids** are not allowed. For example, **ipfw** does not allow "... **uid** tom,dick,harry" or "... **uid** 1000-1002".
- Outbound traffic works in a similar way, just reversing the source and destination.
- Denied traffic will generally have an indication of "<application>: sendto: Permission denied"
- ICMP traffic cannot be reliably filtered using **uid/gid**. This is a known limitation.
- As noted in [ipfw\(8\)](#), some contexts such as initial incoming SYN packets, may have no **uid/gid** associated with them.
- Programs using **setuid(2)** system calls may not behave as expected, though it may be possible to set the **uid/gid** to the effective id if it can be determined.
- Using **uid/gid** keywords for matching is resource intensive and should be used sparingly if at all.

3.4. Lookup Tables

Lookup tables are a versatile feature of many firewall systems, including **ipfw**. A lookup table is a virtual container that holds tuples of elements, one of which is a key that functions as a fast lookup feature. Using a key provided by a rule, **ipfw** can quickly determine if the element is in the table. If it is, that portion of the rule is matched the value associated with the key is used according to the rule.

Lookup tables are a powerful feature of **ipfw** and useful in many situations.

ipfw provides five types of lookup tables:

- **Address tables (addr)** - These tables hold addresses that **ipfw** can rapidly find with an address as a key. If the address is matched the lookup is considered matched and used by the associated rule. This table type takes an additional keyword 'valtype' that can be used to specify IPv4 addresses or IPv6 addresses.

- **Interface tables (iface)** - These tables hold interface names. Each entry is the name of an interface. Note that wildcards such as `em*` are **not** supported.
- **Number tables (number)** - These tables are used for protocols, ports, uids/gids, or jail ids. Entries are 32 bit unsigned integers. Ranges (for example, 1234-5678) are **not** supported.
- **Flow tables (flow)** - These tables contain flow type suboptions that are used in looking up existing traffic flows.
- **MAC** - A MAC address type table holds media access control (MAC) addresses as an address with optional mask length. The mask length defaults to 48 bits if not otherwise specified.

3.4.1. Creating Lookup Tables

All tables must be created before they can be referenced by a rule. Note that the commands to manage tables do not have line numbers - they are independent shell commands that exist outside the ruleset.

```
# ipfw table foo create
```

By default, the **ipfw table create** command creates tables of type **addr**. Table names share the same namespace and so must be unique even among tables of different types.

To specify other types, add the type keyword:

```
# ipfw table bar create type iface
#
# ipfw table baz create type flow
#
# ipfw table bop create type number
```



Previously, when creating a table of type number, a bug existed that required an algorithm option such as `number:array`. This bug was fixed and documented [here](#).

To see all tables, use the **list** subcommand to show the table and any contents. Shown below, all four table types are created and one entry is added to each table:

```
# ipfw table all list
#
# ipfw table foo create type addr
# ipfw table foo add 192.168.1.100 33
added: 192.168.1.100/32 33
#
# ipfw table bar create type iface
# ipfw table bar add em0 33
added: em0 33
#
# ipfw table baz create type number
```

```

# ipfw table baz add 9999 33
added: 9999 33
#
# ipfw table bop create type flow
# ipfw table bop add 203.0.113.10,192.168.1.100 33
ignored: 33
ipfw: Adding record failed: Invalid argument
#
# ipfw table bop destroy
#
# ipfw table bop create type flow:src-ip,dst-ip
# ipfw table bop add 203.0.113.10,192.168.1.100 33
added: 203.0.113.10,192.168.1.100 33
#
# ipfw table bip create type mac
# ipfw table bit add 58:9c:fc:01:02:03 33
added: 58:9c:fc:01:02:03/48 33
#
# ipfw table all list
--- table(bar), set(0) ---
em0 33
--- table(baz), set(0) ---
9999 33
--- table(bip), set(0) ---
58:9c:fc:01:02:03/48 33
--- table(bop), set(0) ---
203.0.113.10,192.168.1.100 33
--- table(foo), set(0) ---
192.168.1.100/32 33

```

Note the error on the flow table above. Flow tables take an explicit flow specification (discussed below) when they are created. When trying to add an entry to a flow table that does not match the flow specification, **ipfw** throws an error.



Note that the output of **ipfw table all list** is ordered by table *name*, not table *type*. This is worth remembering when using many tables of different types. Also, the command does not actually display the table type. Use the command **ipfw table <tablename> info** to display the table type.

In all the examples above a key and a value were added to each table. The key was added according to the table type (addr, iface, etc.) The values above, all set to the integer value 33, are just placeholders in the examples. The *value* for each key is whatever makes sense for the firewall administrator.

"Whatever makes sense" depends on how the table will be used. [ipfw\(8\)](#) identifies some 12 different uses for table values - **skipto** rule number to skip to, **pipe** number, **fib** number, **nat** number to jump to, **dscp** value to match or set, **tag** number to match or set, **divert** port number to divert traffic to, **netgraph** hook number to move packet to, **limit** maximum number of connections, **ipf4** ipv4 next hop to forward packets to, **ipv6** ipv6 next hop to forward packets to, **mark** value to match

or set.

Some of these keywords have already been studied, and will return to assist with table operations.

Tables can be removed one at a time with the **destroy** subcommand, provided the table is not used in any rule:

```
# ipfw table bar destroy
```

or removed all at once by specifying the special name all:

```
# ipfw table all destroy
```



Note that there is *no confirmation* with the **ipfw table all destroy** command as there is with the **ipfw flush** command, so make sure that is the intended action.

A table cannot be destroyed if it is used in a rule.



```
# *ipfw table all destroy*  
ipfw: failed to destroy table(redhosts) in set 0: Device busy
```

Delete the rule using the table, then delete the table.

3.4.2. Using Tables in Rules

To begin using tables in rules, it is first necessary to understand the use of the word **tablearg** which is frequently found in [ipfw\(8\)](#).

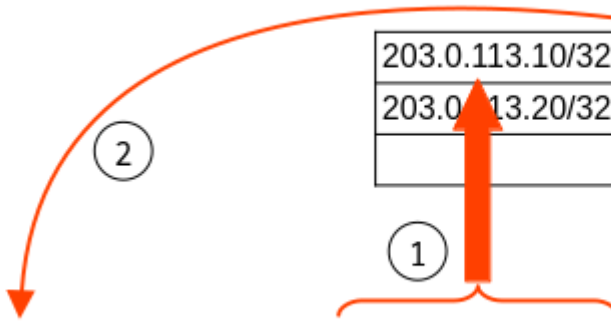
3.4.2.1. Understanding the Word tablearg

A **tablearg** is a value that is the result of a table lookup using a key supplied by a field in a packet. Which field depends on the table type, as discussed above.

The term **tablearg** is used to show where in the rule the retrieved value will be applied.

Table: **badhosts**, type addr

203.0.113.10/32	65535
203.0.113.20/32	65535



```
ipfw add 50 skipto tablearg ip from 'table(badhosts)' to any
```

Figure 12. **tablearg** Keyword Used in a Rule

For a rule with a **tablearg** keyword, **ipfw**

1. Looks up the key in the identified table. The key is supplied by one or more fields in the packet itself.
2. Applies the value associated with that key in the table to replace the word **tablearg**.

Essentially

```
# ipfw add 50 skipto tablearg ip from 'table(badhosts)' to any
```

becomes

```
# ipfw add 50 skipto 65535 ip from 203.0.113.10 to any
```

if the from address in the packet is matched in the table.

If it is not matched, processing continues with the next rule.



From the [ipfw\(8\)](#) man page - The **tablearg** argument can be used with the following actions: **nat**, **pipe**, **queue**, **divert**, **tee**, **netgraph**, **ngtee**, **fwd**, **skipto**, **setfib**; with action parameters: **tag**, **untag**; and with rule options: **limit**, **tagged**.

First Example

The first example is to use a table with the **skipto** keyword.

Consider a table of addresses of "bad hosts". If any such host were to try to connect to or through the firewall, they should be denied. Since there is already a deny rule (the immutable rule at 65535), it is possible to load an address table with keys of hosts, and values of the deny rule, 65535:

First, on the **firewall** VM, create a table called *badhosts*, and populate it with the addresses of hosts to be denied:

```
Restart ipfw:
```

```
# kldunload ipfw
# kldload ipfw
```

Create and populate the table:

```
# ipfw table badhosts create type addr
# ipfw table badhosts add 203.0.113.10/32 65535
added: 203.0.113.10/32 65535
```

Then create a rule that uses the table. For this use case, put the rule *before* the **check-state** rule,

```
# ipfw add 50 skipto tablearg ip from 'table(badhosts)' to any
00050 skipto tablearg ip from table(badhosts) to any
#
# ipfw add 100 check-state
#
# ipfw add 1000 allow ip from any to any
```

Note the single quotes around the `table(badhosts)` entry to placate the shell:

To test, start up **userv3.sh** on the **firewall** VM:

```
# sh userv3.sh
Starting UDP listeners on [5656],[5657],[5658]
```

And on the **external1** VM (which should have address 203.0.113.10) , start up **ucon.sh**:

```
# sh ucon.sh 5656
UDP communicationg [203.0.113.50],[5656],[1]
```

No communication should be seen on the **firewall** VM. The **skipto** rule matched the address in the table, and the **tablearg** keyword was replaced with the default deny rule. This can be verified by reviewing the counters with **ipfw show**.

However, by removing the entry for `203.0.113.10/32` from the `badhosts` table, the communications succeed and reach the **userv3.sh** services listening on the **firewall** VM:

```
# ipfw table badhosts delete 203.0.113.10
deleted: 203.0.113.10/32 0
```

Retrying the **ucon.sh** communications above will succeed.

It would be tempting to combine the previous two examples into something like:

```
# ipfw add 1000 allow udp from 'table(badhosts)' to me dst-port 'table(badports)'  
ipfw: invalid destination port table(badports)
```

but **ipfw** does not allow the use of more than one table in a rule.

However, instead of a second **table** keyword, it is possible to use the **lookup** keyword for the port:

A contrived example:

```
# ipfw add 25 allow udp from 'table(badhosts)' to me lookup dst-port badports  
00025 allow udp from table(badhosts) to me dst-ip lookup dst-port badports
```

While this does work, the better solution is to use the a **flow** table:

Unload and reload the **ipfw** kernel module for the next example, this time for "good hosts".

```
# ipfw table goodflow create type flow:src-ip,dst-port  
#  
# ipfw table goodflow add 203.0.113.10,5656  
added: 203.0.113.10,5656 0  
#  
# ipfw table goodflow add 203.0.113.10,5657  
added: 203.0.113.10,5657 0  
#  
# ipfw add 500 allow udp from any to me flow 'table(goodflow)'  
00500 allow udp from any to me flow table(goodflow)
```

This gives the firewall admin much more granular control of exactly what host and what port to match together. Startup the **userv3.sh** script on the **firewall** VM and note the results by trying **sh ucon.sh 5656**, **sh ucon.sh 5657**, and **sh ucon.sh 5658** from the **external1** VM. The first two succeed, while the third does not.

Using the **valtype** keyword for addr tables permits separate tables for IPv4 and IPv6:

```
# Create the translation tables.  
# ipfw table T46 create type addr valtype ipv6  
# ipfw table T64 create type addr valtype ipv4
```

Second Example

The second example concerns using tables with the **limit** keyword. Recall that the **limit** keyword limits the number of active connections at one time.

Reset **ipfw** by unloading and loading the **ipfw.ko** kernel module.

Now, consider an address table to keep track of addresses and limits like this:


```
# Create a table named "limits".
# ipfw table limits create type addr valtype limit
#
# Assign a value of 23 to the address 203.0.113.10
# ipfw table limits add 203.0.113.10 23
added: 203.0.113.10/32 23
#
# Now add a limit rule for this address
# ipfw add 1000 allow udp from 'table(limits)' to me limit src-addr tablearg
01000 allow udp from table(limits) to me limit src-addr tablearg :default
#
# And add a rule to allow the traffic.
# ipfw add 1100 allow udp from 'table(limits)' to me
#
```

The above rules have created a table named "limits" of tabletype *addr* and value type *limit* with one entry, 203.0.113.10 with value 23. With rule 1000, all packets coming into the firewall will be looked up in the table. If a packet from address 203.0.113.10 arrives, the lookup will succeed and the value of "23" will be applied as the tablearg to the **limit** option for connections with that address.

Once this is set up, it is possible to change the limit value without changing the rule. The value can either be changed by deleting and re-adding the table entry.

While this looks like a good solution, **there is a bug here**.

ipfw has failed to correctly initialize the value in the table.

Listing the table shows the problem:

```
# ipfw table limits list
203.0.113.10/32 0
```

This has been logged as [Bug 284691](#) and will be tracked as this book goes to publication.

3.4.2.2. More on flow tables

A **flow** table maintains a list of **flows**. A **flow** is a designation given to traffic between two endpoints. The designation can be any subset of:

- **src-ip**
- **src-port**
- **proto**
- **dst-ip**
- **dst-port**

that makes sense in source → destination order. Examples include:

```
# ipfw table zoo create type flow:src-ip # Creates a flow based on source IP address
# ipfw table zar create type flow:proto # Creates a flow based on just a protocol
# ipfw table zaz create type flow:dst-ip # Creates a flow based on just the
destination IP
# ipfw table zop create type flow:dst-port # Creates a flow based on the destination
port
```

or with extra specificity:

```
# ipfw table zip create type flow:src-ip,proto,dst-ip,dst-port # Flow based on all
four
# ipfw table zim create type flow:src-ip,proto # Just source IP and protocol
# ipfw table zam create type flow:src-ip,proto,dst-ip # Source IP, destination IP and
protocol
# ipfw table zap create type flow:src-ip,dst-ip # Just IP address endpoints
```

Once the table is created for a **flow**, entries can be placed in the table provided they match the table **flow** specification.

Matching these additions to the tables created above:

```
# ipfw table zim add 192.168.200.30,tcp
```

would succeed, but

```
# ipfw table zip add 192.168.30.20,172.20.15.20
```

would fail because the flow specification for table zip is **src-ip,proto,dst-ip,dst-port** and neither the protocol nor the destination port was given.

A correct flow specification for table zip would be something like:

```
# ipfw table zip add 192.168.30.20,tcp,172.20.15.20,80
```

flow tables allow for precise definition of traffic between a source and a destination. Once in the table, rules can be applied to commands **allow**, **deny**, **divert**, **queue**, etc. for modifying traffic flow.

General notes on all tables:

- Table names can be numeric or alphanumeric and can include only hyphen (-), underscore (_), and period (.) as special characters.
- Maximum table name length is 63 characters.
- Tables names must be unique within a set. Tables can have the same name across different sets, however any rules for tables in sets other than set 0, must include the set number. The `sysctl`

variable `net.inet.ip.fw.tables_sets` controls this behavior.

- The maximum number of tables across all sets is 65535. Practically however, the number is controlled by the sysctl variable `net.inet.ip.fw.tables_max`. The default is 128.
- If a table is in use in a rule, it cannot be destroyed. The rule must be removed first, then the table can be destroyed. However, a table can be flushed (**`ipfw table tablename flush`**) at anytime.
- All table types survive an **`ipfw flush`** action, and table contents are not affected.
- Make table names as descriptive as possible to avoid confusion when used in rules. The names used here (foo, bar, zip, zap, etc.) are just examples.
- As noted in the man page, if two tables are used in a rule, the result of the second (destination) is used. Therefore avoid using two tables in a rule, or try using the **`lookup`** keyword instead.

Notes on specific table types:

- **Address tables (addr)**

- Address tables (addr) support IPv4 and IPv6 address types, and varying mask lengths appropriate for each address type. The default mask length for IPv4 is 32 bits (/32) and the default prefix for IPv6 is 128 bits (/128).
- Table lookups will return the most specific entry, so 203.0.113.20/32 is preferred over 203.0.113.0/24.

- **Interface tables (iface)**

- Interface tables store interface names as alphanumeric text. The text does not actually have to match a current valid interface.
- Special characters in interface names can include any from the set

```
[ -+ _ ? , . ! ~ @ # % ^ & * ( ) = ; : / < > { } | ]
```

Note however, that the shell may recognize some of these characters when adding and during lookup, thus interfering with the table operation, and so *special characters in interface names should be avoided*.

- The maximum key length is 15 characters. You can create an entry with a longer name and **`ipfw`** will not throw an error, but the entry will be truncated to 15 characters.
- It is possible to add interface names consisting of Unicode characters, though support for Unicode character sets in user terminals varies. For example, to add the interface named "meλ" ("em ee lambda") with value 32 to table "myintf" you would type:

```
# ipfw table myintf add emCtl+Shift+U 03BB 32
```

Note however, that it may not be possible to actually create the named interface.

- There is **no support for interface ranges**, for example `em0-4`, even though an interface name "**`em0-4`**" can be entered.

- **Number tables (number)**
 - Number tables support unsigned 32-bit integer types.
 - Entries can be positive or negative. Negative entries perform unsigned ones complement arithmetic, and positive numbers roll over from 4294967295 to 0.
 - Any shell element that evaluates to a number can be used: shell variables that resolve to a number (`$MAILCHECK`, `$PPID`), backtick operations such as `expr 5 + $UID`, `id -u`, `date +%s`, and special variables like `$RANDOM` in bash.
 - As with other table types, ranges are not supported.
- **Flow tables (flow)**
 - Flow tables describe network traffic based on the desired attributes. The best matches include as much detail as possible: `src-ip`, `proto`, `dst-ip`, `port`. Including less than that may make it difficult to add an element in the table:
 - `ipfw table foo create type flow:dst-port` # Table based on just the destination port
 - `ipfw table foo add telnet` # Fails to add!
 - `ipfw table bar create type flow:dst-ip,dst-port`
 - `ipfw table bar add 203.0.113.10,5656`

3.5. Stream Control Transport Protocol (SCTP)

A readable introduction to SCTP is found in Wikipedia - https://en.wikipedia.org/wiki/Stream_Control_Transmission_Protocol#RFCs and ever more detail is found in the accompanying RFCs.

3.5.1. SCTP Versions

As of FreeBSD 14.1, there are three different versions of SCTP - a "native" version, a "separate portable" version, and a "userland" version.

- FreeBSD ships with the "native" version of the protocol, described in [sctp\(4\)](#). (This is actually the reference implementation of SCTP initially developed on FreeBSD 7.) This version requires loading the kernel module `sctp.ko` before use. Native SCTP uses the SCTP library on FreeBSD which provides access to the various functions found in `<netinet/sctp.h>`. Developers can use the features described in [sctp\(4\)](#) to develop SCTP applications based on the native protocol.

Then, there are a couple of notable packages regarding SCTP:

- **libusrctp** “Portable SCTP userland stack” – This is a non-kernel implementation of the protocol using “usrctp_*” functions. It also provides UDP encapsulation as shown in the Wireshark image below:
- **sctplib** “User-space implementation of the SCTP protocol RFC 4960” – This package is a re-implementation of the native SCTP code that can function as a replacement for the default installed code. It does not require the `sctp.ko` kernel module.



As noted in the **BUGS** section of [sctp\(4\)](#), the `sctp.ko` kernel module cannot be

unloaded. Restart FreeBSD to remove the module from the kernel.

Interestingly, the reference implementation for SCTP that was developed on FreeBSD 7 and has been archived on GitHub at <https://github.com/cyberroadie/sctp-examples>. **ipfw** played a role in its development and the vestiges of that development remain in the **ipfw** code base and in the collection of **sysctls** that support it.

The following section studies the native SCTP protocol usage and the encapsulated usage with **ipfw**.

3.5.2. SCTP Protocol Operation

The image below shows a Wireshark view of a typical native SCTP association (connection):

No.	Time	Source	Destination	Protocol	Length	Info
3	0.001307916	203.0.113.10	10.10.10.20	SCTP	146	INIT
4	0.006924328	10.10.10.20	203.0.113.10	SCTP	466	INIT_ACK
5	0.007561295	203.0.113.10	10.10.10.20	SCTP	374	COOKIE_ECHO
6	0.009417648	10.10.10.20	203.0.113.10	SCTP	50	COOKIE_ACK
7	0.012005419	203.0.113.10	10.10.10.20	SCTP	1062	DATA (TSN=0)
8	0.013430013	203.0.113.10	10.10.10.20	SCTP	1062	DATA (TSN=1)
9	0.014416103	10.10.10.20	203.0.113.10	SCTP	62	SACK (Ack=0, Arwnd=1862879)
10	0.015598441	10.10.10.20	203.0.113.10	SCTP	62	SACK (Ack=1, Arwnd=1862879)
11	0.016136268	203.0.113.10	10.10.10.20	SCTP	54	SHUTDOWN
12	0.017325772	10.10.10.20	203.0.113.10	SCTP	50	SHUTDOWN_ACK
13	0.017759339	203.0.113.10	10.10.10.20	SCTP	50	SHUTDOWN_COMPLETE

Figure 13. Wireshark View of Native SCTP

Notice how there are two different interfaces involved with this transfer - 127.0.0.1, and 192.168.1.78. In fact, SCTP supports association setup and data exchange with multiple interfaces to a remote node. The RFC's explain this capability in detail.

Also in this image, note the basic **4-way handshake** - INIT, INIT_ACK, COOKIE_ECHO, and COOKIE_ACK. Also shown are the HEARTBEAT, DATA, SACK (Stream Acknowledgement) messages, and the shutdown sequence of SHUTDOWN, SHUTDOWN_ACK, and SHUTDOWN_COMPLETE.

Below is a view of **netstat -an** showing the display of a separate SCTP section containing all current associations:

```
% netstat -an
Active Internet connections (including servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         (state)
tcp4      0      0 *.22                    *.*                     LISTEN
tcp6      0      0 *.22                    *.*                     LISTEN
tcp4      0      0 127.0.0.1.631          *.*                     LISTEN
tcp6      0      0 ::1.631                 *.*                     LISTEN
udp4      0      0 127.0.0.1.123          *.*                     LISTEN
udp6      0      0 fe80::1%lo0.123        *.*                     LISTEN
udp6      0      0 ::1.123                 *.*                     LISTEN
udp4      0      0 192.168.1.78.123       *.*                     LISTEN
udp6      0      0 2600:1700:3901:4.123   *.*                     LISTEN
udp6      0      0 fe80::3e97:eff:f.123   *.*                     LISTEN
udp4      0      0 *.123                    *.*                     LISTEN
```

```

udp6      0      0 *.123          *.*
Active SCTP associations (including servers)
Proto Type Local Address Foreign Address (state)
sctp4 1to1 127.0.0.1.5000 127.0.0.1.42227 ESTABLISHED
          192.168.1.78.5000 192.168.1.78.42227
sctp4 1to1 127.0.0.1.42227 127.0.0.1.5000 ESTABLISHED
          192.168.1.78.42227 192.168.1.78.5000
sctp4 1to1 127.0.0.1.5000 LISTEN
          192.168.1.78.5000
Active UNIX domain sockets
Address Type Recv-Q Send-Q Inode Conn
Refs Nextref Addr
fffff8015575c000 stream 0 0 0 fffff80155758c00
0 0
. . .

```

Internally, the SCTP data packets look different from TCP and/or UDP packets. However, like those two, SCTP has a protocol number (hexadecimal 0x84, decimal 132), source address, destination address, and port numbers, so using **ipfw** with SCTP will be fairly straightforward.

Here’s a look at a typical packet:

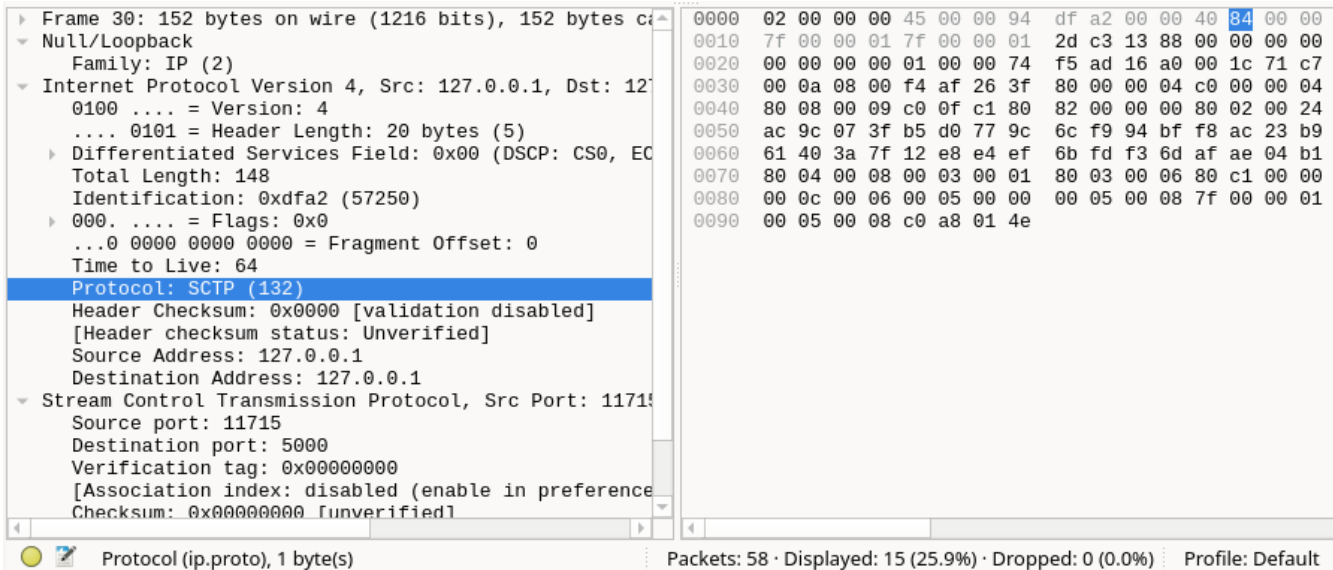


Figure 14. Internal View of the SCTP INIT Data Packet

To study SCTP’s behavior with **ipfw** the client/server echo program, shown above, will be used, along with a separate streaming application similar to the familiar *chargen* small server.

Also needed is the **internal** VM. If this VM has not yet been set up, do so now following the general guidelines of the [Qemu Setup](#) section for details. Be sure to install all the required packages, including **tsctp**.

3.5.3. Using the TSCTP Testing Tool on FreeBSD

A handy testing tool is the FreeBSD package **tsctp**. If this package was not previously installed, install the package on the **external1** and **internal** VMs. (Configure the VMs to access the Internet to

install the package. Refer to the [Qemu Setup](#) section for details.)

```
# pkg install tsctp
```

This tool uses the native SCTP protocol version. To use, first load the **sctp.ko** kernel module on the **external1** and **internal** VMs:

```
# kldload sctp.ko
```

Running the program is discussed below.

SCTP Test Configuration

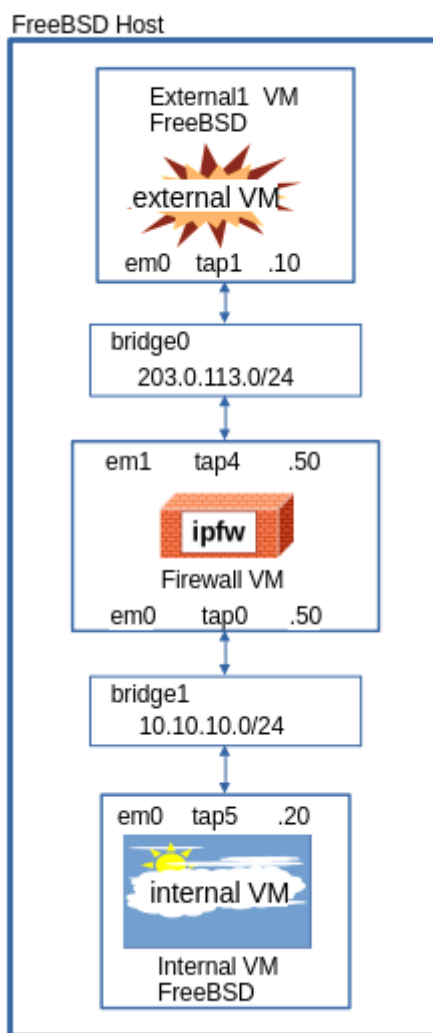


Figure 15. Setting Up to Test Native SCTP

To get started, on the FreeBSD host machine, initialize the **bridge** and **tap** devices to the architecture shown in the figure above.

```
% cd ~/ipfw-primer/ipfw/HOST_SCRIPTS
% sudo /bin/sh mkbr.sh reset bridge0 tap1 tap4 bridge1 tap0 tap5
```

Then start the required VMs:

```
% /bin/sh runvm.sh firewall external1 internal
```

Then follow these steps:

1. Apply the addressing shown in the above figure and ensure connectivity with adjacent interfaces, and with the opposite side interfaces (10.10.10.20 from the **external1** VM and 203.0.113.10 from the **internal** VM).
2. The default route for the **external1** VM should point to 203.0.113.50, and for the **internal** VM it should point to 10.10.10.50.
3. Load the **sctp.ko** kernel module on the **external1** and the **internal** VMs. This enables communication via native SCTP on these VMs.

On the **internal** VM, run in server mode:

```
# tsctp -L 127.0.0.1 -L 10.10.10.20 -p 1234
```

and on the **external1** VM, run in client mode:

```
# tsctp -L 203.0.113.10 -p 1234 -n 10 -l 1000 -V 10.10.10.20
```

Use `tsctp --help` to get a list of the options and meanings.

The `-V` option will print a list of messages being send by the client.

The client program sends 10 messages (`-n 10`) to the server. Simple statistics are shown on both the client and the server once the program terminates.

The above procedure has established SCTP communications across the firewall. The **firewall** VM does not need to load the **sctp.ko** module. To the **firewall** VM, this is simply normal IP traffic.

Restart the client with `-n 0`, for unlimited messages. While normal SCTP traffic is established from client to server, load **ipfw** on the **firewall** VM.

Traffic is immediately halted. Eventually, the client will recognize it has been disconnected. This may take a couple of minutes. The disconnection is shown in the image below:


```

root@external1:~ # tsctp -L 127.0.0.1 -L :::1 -L 203.0.113.10 -p 1234 -n 10 -l 1000 10.10.10.20
Sending of 10 messages of length 1000 took 0.014762 seconds.
Throughput was 677414.984419 Byte/sec.
root@external1:~ #
root@external1:~ #
root@external1:~ # tsctp -L 127.0.0.1 -L :::1 -L 203.0.113.10 -p 1234 -n 0 -l 1000 10.10.10.20
sctp_sendmsg: Connection reset by peer
sctp_sendmsg: No such file or directory
Sending of 24648 messages of length 1000 took 249.540962 seconds.
Throughput was 98773.362908 Byte/sec.
root@external1:~ # █

```

Figure 16. IPFW Firewall Disrupts SCTP Traffic

Creating suitable rules for SCTP traffic is quite similar to other rules performed in previous examples.

```

root@firewall:~ # ipfw add 100 check-state
00100 check-state :default
root@firewall:~ #
root@firewall:~ # ipfw add 1000 allow sctp from 203.0.113.10 to 10.10.10.20
01000 allow sctp from 203.0.113.10 to 10.10.10.20
root@firewall:~ # ipfw add 2000 allow sctp from 10.10.10.20 to 203.0.113.10
02000 allow sctp from 10.10.10.20 to 203.0.113.10
root@firewall:~ #
root@firewall:~ # ipfw zero
Accounting cleared.
root@firewall:~ # ipfw zero 65535
Entry 65535 cleared.
root@firewall:~ #
root@firewall:~ # ipfw show
00100 0 0 check-state :default
01000 0 0 allow sctp from 203.0.113.10 to 10.10.10.20
02000 0 0 allow sctp from 10.10.10.20 to 203.0.113.10
65535 0 0 deny ip from any to any
root@firewall:~ #

```

One would think the "one-rule" version using **setup** and **keep-state** keywords would work, but it does not. The "two-rule" version must be used.

Also, keep in mind that SCTP is typically used where there are multiple interfaces per association. This example has used only one, but the principles are the same.

3.5.4. Downloading and Building `usrstcp` Programs

To test SCTP encapsulation with UDP, download and build the **usrstcp** kit. Follow this procedure to download and build the programs for the **usrstcp** kit. On the **external1** VM and the **internal** VM, the procedure is the same.

For this procedure, reconfigure the **external1** and **internal** VMs to access the Internet as described in [Appendix A](#). Then perform these steps:

```
# pkg install git
# pkg install cmake
# mkdir /root/src
# cd /root/src
# git clone https://github.com/sctplab/usrctp.git
# mkdir tmp
# cd tmp
# cmake ../usrctp
# cmake --build .
```

Once finished, the test programs are located in `/root/src/tmp/programs`.

Now reconfigure **external1** and **internal** VMs back to the architecture and addressing (including default routes) in [Using the TSCTP Testing Tool](#).

3.5.5. Encapsulated Echo Server and Client with IPFW

The figure below shows encapsulated usage of SCTP with the **chargen_server_upcall** program running on the **internal** VM and the **client_upcall** program running on the **external1** VM. The output is similar to that of the *chargen* small server program.



If the client doesn't start right away, hit `Enter` one time.

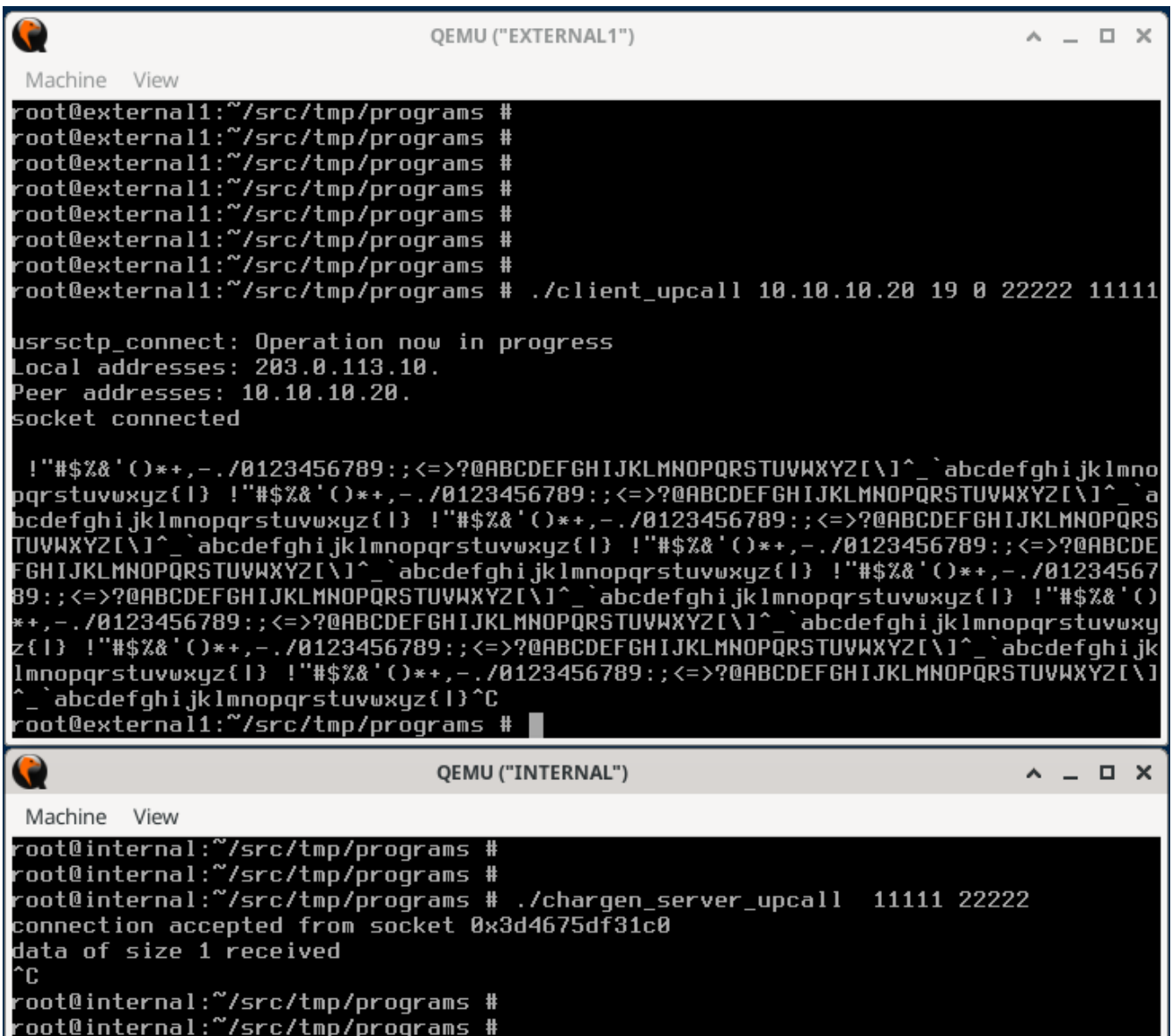


Figure 17. SCTP Traffic Encapsulated in UDP Datagrams

All the data exchanged between the two systems was encapsulated in UDP datagrams as shown in the figure below.

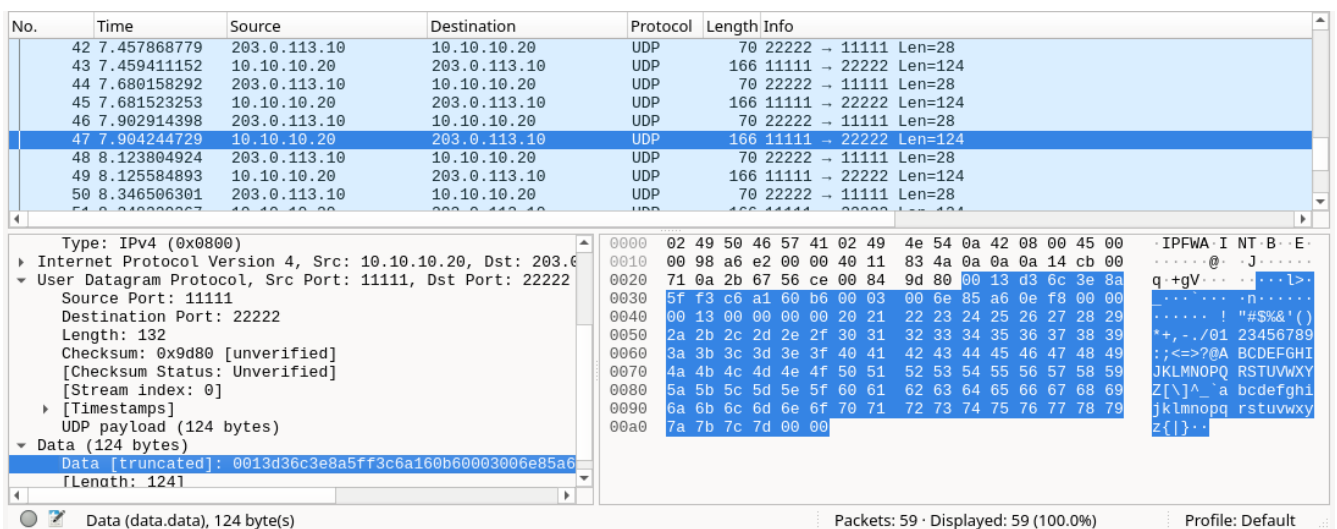


Figure 18. Wireshark View of UDP Encapsulation of SCTP

Any **ipfw** rules for this traffic only have to be concerned with UDP, not SCTP.

Chapter 4. IPFW Dummynet and Traffic Shaping

FreeBSD's **dummynet** is not a network for dummies. It is a sophisticated network traffic shaping tool for bandwidth usage and scheduling algorithms. In this use of **ipfw**, the focus is not on ruleset development, although rules are still used to select traffic to pass to **dummynet** objects. Instead, the focus is on setting up a system to shape traffic flows. **dummynet** provides tools to model scheduling, queuing, and similar tasks similar to the real-world Internet.

dummynet works with three main types of objects - a **pipe**, a **queue**, and a **sched** (short for scheduler) which also happen to be the three keywords to now examine.

A **pipe** (*not to be confused with a Unix pipe(2)!*) is a model of a network link with a configurable bandwidth, and propagation delay.

A **queue** is an abstraction used to implement packet scheduling using one of several different scheduling algorithms. Packets sent to a queue are first grouped into flows according to a mask on a 5-tuple (protocol, source address, source port, destination address, destination port) specification. Flows are then passed to the scheduler associated with the queue, and each flow uses scheduling parameters (weight, bandwidth, etc.) as configured in the queue itself. A **sched** (scheduler) in turn is connected to a **pipe** (an emulated link) and arbitrates the link's bandwidth among backlogged flows according to weights and to the features of the scheduling algorithm in use.

Network performance testing is a complex subject that can encompass many variables across many different testing strategies. To understand the basics behind **dummynet**, it is not necessary to dive into the deepest levels of network performance testing - only enough to understand how to use **dummynet**. Also, these tests are restricted to using IP and TCP exclusively.

Setting Up for Traffic Measurement

Most of the examples in this section can be done with the architecture used in the [original lab setup](#) in Chapter 2, copied here for reference:

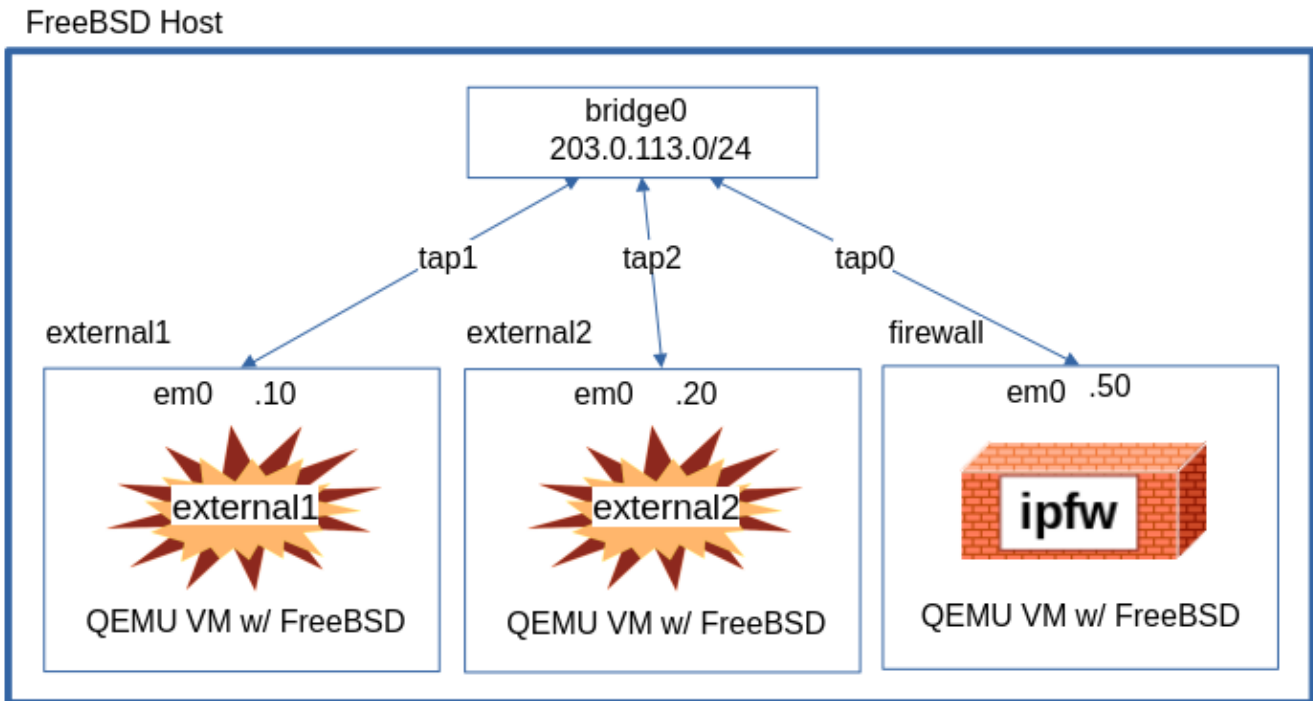


Figure 19. IPFW Lab for **dummynet** Examples

Use this **bridge** and **tap** configuration on the FreeBSD host system:

```
% sudo /bin/sh mkbr.sh reset bridge0 tap0 tap1 tap2
% /bin/sh swim.sh # or scim.sh for screen(1)
% /bin/sh runvm.sh firewall external1 external2
```

Apply the correct addressing for each VM in this example.

Where necessary, additional virtual machines can be created and added to the bridge.

4.1. Measuring Default Throughput

The idea behind **dummynet** is that it lets one model and/or shape network speeds, available bandwidth, and scheduling algorithms. But it is first necessary to know what the current transfer speeds are for the current environment (QEMU virtual machines over a FreeBSD bridge). To find out, here is a short detour to learn **iperf3**, the network bandwidth testing tool used to perform simple transfer and bitrate calculations.

iperf3, can determine the effective throughput of data transfer for a network. Sometimes called "goodput", this is the basic speed the user sees for transferring data across the network - the value that is unencumbered by protocol type and overhead.

To use **iperf3**, ensure that the software is installed on both the **firewall** VM system, and the **external1** VM (and **external2** and **external3**), and that **ipfw** on the **firewall** VM is disabled (**# kldunload ipfw**).

The basic operation of **iperf3** is as a client-server architecture, so on the **external1** VM system, start the **iperf3** software in server mode:

```
# iperf3 -s <--- run iperf3 in server mode
-----
Server listening on 5201 (test #1)
-----
. . .
```

Then, on the **firewall** VM, run the client:

```
# iperf3 -c 203.0.113.10 <--- connect to external1 server and send test data
Connecting to host 203.0.113.10, port 5201
[ 5] local 203.0.113.50 port 19359 connected to 203.0.113.10 port 5201
[ ID] Interval          Transfer      Bitrate      Retr  Cwnd
[ 5]  0.00-1.03    sec  12.5 MBytes  102 Mbits/sec  0    1.07 MBytes
[ 5]  1.03-2.09    sec  13.8 MBytes  108 Mbits/sec  0    1.07 MBytes
[ 5]  2.09-3.07    sec  12.5 MBytes  107 Mbits/sec  0    1.07 MBytes
[ 5]  3.07-4.09    sec  12.5 MBytes  103 Mbits/sec  0    1.07 MBytes
[ 5]  4.09-5.08    sec  12.5 MBytes  106 Mbits/sec  0    1.07 MBytes
[ 5]  5.08-6.09    sec  12.5 MBytes  105 Mbits/sec  0    1.07 MBytes
[ 5]  6.09-7.07    sec  12.5 MBytes  107 Mbits/sec  0    1.07 MBytes
[ 5]  7.07-8.05    sec  12.5 MBytes  107 Mbits/sec  0    1.07 MBytes
[ 5]  8.05-9.04    sec  12.5 MBytes  106 Mbits/sec  0    1.07 MBytes
[ 5]  9.04-10.02   sec  12.5 MBytes  107 Mbits/sec  0    1.07 MBytes
-----
[ ID] Interval          Transfer      Bitrate      Retr
[ 5]  0.00-10.02   sec  126 MBytes  106 Mbits/sec  0
[ 5]  0.00-10.02   sec  126 MBytes  106 Mbits/sec
                                     sender
                                     receiver

iperf Done.
#
```

A key test for measuring throughput is to send a file of data and measure the transfer speed. To create the file, use [jot\(1\)](#) on the **firewall** VM:

```
# jot -r -s "" 10000000 0 9 > A.bin
```

This command creates a file of random ASCII digits exactly 10,000,001 bytes long. (Note that it can take anywhere from 30 seconds to two minutes to create the file on a QEMU virtual machine.)

To transfer the file to the server on the **external1** VM use this command:

```
# iperf3 -F A.bin -c 203.0.113.10 -t 10
Connecting to host 203.0.113.10, port 5201
[ 5] local 203.0.113.50 port 51657 connected to 203.0.113.10 port 5201
[ ID] Interval          Transfer      Bitrate      Retr  Cwnd
[ 5]  0.00-1.04    sec  12.5 MBytes  101 Mbits/sec  0    490 KBytes
[ 5]  1.04-1.52    sec   5.81 MBytes  101 Mbits/sec  0    490 KBytes
-----
```

```

[ ID] Interval          Transfer    Bitrate      Retr
[ 5]  0.00-1.52   sec  18.3 MBytes  101 Mbits/sec    0          sender
      Sent 18.3 MByte / 18.3 MByte (100%) of A.bin
[ 5]  0.00-1.52   sec  18.3 MBytes  101 Mbits/sec          receiver
iperf Done.
#

```

Running this command several times shows that a consistent average bitrate for throughput on this system is about 101Mbits/second - or about 18.3 MBytes/second. (Your values will differ on your local machine.)

There is now a baseline TCP-based "goodput" value for testing **dummysnet** traffic shaping commands.

4.2. IPFW Commands for Dummysnet

To use **dummysnet**, load the kernel module **dummysnet.ko** in addition to the **ipfw.ko** module on the **firewall** VM:

```

# kldload ipfw
# kldload dummysnet
load_dn_sched dn_sched FIFO loaded
load_dn_sched dn_sched QFQ loaded
load_dn_sched dn_sched RR loaded
load_dn_sched dn_sched WF2Q+ loaded
load_dn_sched dn_sched PRIQ loaded
load_dn_sched dn_sched FQ_CODEL loaded
load_dn_sched dn_sched FQ_PIE loaded
load_dn_aqm dn_aqm CODEL loaded
load_dn_aqm dn_aqm PIE loaded
#

```

dummysnet announces the schedulers it is configured to use.

4.2.1. Simple Pipe Configuration

Recall that **dummysnet** uses **pipes**, **queues**, and **sched** (schedulers) to shape traffic.

To see **dummysnet** in action, create a **pipe** with limited bandwidth, and assign it to a rule matching traffic to the **external1** VM:

```

# Load the ipfw kernel module if needed:
# kldload ipfw
ipfw2 (+ipv6) initialized, divert loadable, nat loadable, default to deny, logging
disabled
#
# ipfw pipe 1 config bw 300Kbit/s
# ipfw pipe 1 show

```



```
00001: 300.000 Kbit/s    0 ms burst 0
q131073 50 sl. 0 flows (1 buckets) sched 65537 weight 0 lmax 0 pri 0 droptail
sched 65537 type FIFO flags 0x0 0 buckets 0 active
#
```

The above output shows the **pipe** configuration limiting bandwidth (bw) to 300Kbits/sec.



Recent versions of FreeBSD now use the command alias **dnctl** for configuration of pipes, queues, and schedulers. See [dnctl\(8\)](#) for details.

Now add **ipfw** rules to send traffic between the **firewall** VM and the **external1** VM through the **pipe**:

```
# ipfw add 100 check-state
00100 check-state :default
#
# ipfw add 1000 pipe 1 ip from any to any
01000 pipe 1 ip from any to any
#
# ipfw list
00100 check-state :default
01000 pipe 1 ip from any to any
65535 deny ip from any to any
#
```

By adding the matching phrase "**ip from any to any**" and assigning it to **pipe 1**, the **firewall** VM is directed to send all ip-based traffic through **pipe 1**, now configured as a 300K bps link.

By re-running the basic file transfer command for **iperf3** these difference take shape:

```
# iperf3 -F A.bin -c 203.0.113.10 -t 10 --length 1460
Connecting to host 203.0.113.10, port 5201
[ 5] local 203.0.113.50 port 39558 connected to 203.0.113.10 port 5201
[ ID] Interval           Transfer     Bitrate      Retr  Cwnd
[ 5] 0.00-1.01    sec   75.6 KBytes  612 Kbits/sec    0   25.6 KBytes
[ 5] 1.01-2.01    sec   41.3 KBytes  339 Kbits/sec    0   51.0 KBytes
[ 5] 2.01-3.01    sec   45.6 KBytes  374 Kbits/sec    0   62.3 KBytes
[ 5] 3.01-4.01    sec   27.1 KBytes  222 Kbits/sec    0   66.6 KBytes
[ 5] 4.01-5.01    sec   35.6 KBytes  292 Kbits/sec    0   66.6 KBytes
[ 5] 5.01-6.01    sec   44.2 KBytes  362 Kbits/sec    0   66.6 KBytes
[ 5] 6.01-7.01    sec   21.4 KBytes  175 Kbits/sec    0   66.6 KBytes
[ 5] 7.01-8.01    sec   37.1 KBytes  304 Kbits/sec    0   66.6 KBytes
[ 5] 8.01-9.01    sec   48.5 KBytes  397 Kbits/sec    0   66.6 KBytes
[ 5] 9.01-10.01   sec   22.8 KBytes  187 Kbits/sec    0   66.6 KBytes
-----
[ ID] Interval           Transfer     Bitrate      Retr
```

```
[ 5] 0.00-10.01 sec 399 KBytes 327 Kbits/sec 0 sender
      Sent 399 KByte / 9.54 MByte (4%) of A.bin
[ 5] 0.00-10.73 sec 379 KBytes 289 Kbits/sec receiver

iperf Done.
```



If the system returns the error "iperf3: error - control socket has closed unexpectedly", simply re-run the command.

Here, during **iperf3's** 10-second run, the **ipfw dummynet** configuration limited the transfer speed to an average of about 327 Kbits/sec, and only about 4% of the entire 10MB file was transferred.

To see how to use **dummynet** to configure different link speeds, set up a second **pipe**:

```
# ipfw pipe 2 config bw 3Mbit/s
# ipfw pipe show
00001: 300.000 Kbit/s 0 ms burst 0
q131073 50 sl. 0 flows (1 buckets) sched 65537 weight 0 lmax 0 pri 0 droptail
      sched 65537 type FIFO flags 0x0 0 buckets 0 active
00002: 3.000 Mbit/s 0 ms burst 0
q131074 50 sl. 0 flows (1 buckets) sched 65538 weight 0 lmax 0 pri 0 droptail
      sched 65538 type FIFO flags 0x0 0 buckets 0 active
#
```

This **pipe** is set up to be 10 times faster (3Mb/sec instead of 300Kb/sec) than **pipe 1**. To test this pipe, start up the **external2** VM and run **iperf3 -s**. Then reconfigure the **ipfw** rules to send to the **external2** VM through **pipe 2**:

```
# ipfw list
00100 check-state :default
01000 pipe 1 ip from any to any
65535 deny ip from any to any
#
# ipfw delete 1000
#
# ipfw add 1000 pipe 1 ip from me to 203.0.113.10 // external1
01000 pipe 1 ip from me to 203.0.113.10
#
# ipfw add 1100 pipe 1 ip from 203.0.113.10 to me // external1
01100 pipe 1 ip from 203.0.113.10 to me
#
# ipfw add 2000 pipe 2 ip from me to 203.0.113.20 // external2
02000 pipe 2 ip from me to 203.0.113.20
#
# ipfw add 2100 pipe 2 ip from 203.0.113.20 to me // external2
02100 pipe 2 ip from 203.0.113.20 to me
#
# ipfw list
```

```

00100 check-state :default
01000 pipe 1 ip from me to 203.0.113.10
01100 pipe 1 ip from 203.0.113.10 to me
02000 pipe 2 ip from me to 203.0.113.20
02100 pipe 2 ip from 203.0.113.20 to me
65535 deny ip from any to any
#

```

As expected, **pipe 2** is approximately 10 times faster than **pipe 1**:

```

# iperf3 -F A.bin -c 203.0.113.20 -t 10 --length 1460
Connecting to host 203.0.113.20, port 5201
[ 5] local 203.0.113.50 port 48108 connected to 203.0.113.20 port 5201
[ ID] Interval           Transfer     Bitrate      Retr  Cwnd
[ 5]  0.00-1.01    sec   399 KBytes  3.24 Mbits/sec    0   64.0 KBytes
[ 5]  1.01-2.01    sec   358 KBytes  2.93 Mbits/sec    0   64.0 KBytes
[ 5]  2.01-3.01    sec   359 KBytes  2.94 Mbits/sec    0   64.0 KBytes
[ 5]  3.01-4.01    sec   364 KBytes  2.98 Mbits/sec    0   64.0 KBytes
[ 5]  4.01-5.01    sec   368 KBytes  3.01 Mbits/sec    0   66.9 KBytes
[ 5]  5.01-6.01    sec   332 KBytes  2.72 Mbits/sec    0   66.9 KBytes
[ 5]  6.01-7.01    sec   362 KBytes  2.97 Mbits/sec    0   66.9 KBytes
[ 5]  7.01-8.01    sec   355 KBytes  2.91 Mbits/sec    0   66.9 KBytes
[ 5]  8.01-9.01    sec   345 KBytes  2.83 Mbits/sec    0   66.9 KBytes
[ 5]  9.01-10.01   sec   344 KBytes  2.81 Mbits/sec    0   66.9 KBytes
-----
[ ID] Interval           Transfer     Bitrate      Retr
[ 5]  0.00-10.01   sec  3.50 MBytes  2.93 Mbits/sec    0          sender
      Sent 3.50 MByte / 9.54 MByte (36%) of A.bin
[ 5]  0.00-10.06   sec  3.48 MBytes  2.90 Mbits/sec          receiver

iperf Done.

```

Next, change the **pipe** configuration without changing the ruleset. Below, the **pipe 1** bandwidth is changed to the equivalent of a telecommunications T1 line as in the days of old:

```

# ipfw pipe 1 config bw 1544Kbit/s
# ipfw pipe show
00001: 1.544 Mbit/s    0 ms burst 0
q131073 50 sl. 0 flows (1 buckets) sched 65537 weight 0 lmax 0 pri 0 droptail
  sched 65537 type FIFO flags 0x0 0 buckets 0 active
00002: 3.000 Mbit/s    0 ms burst 0
q131074 50 sl. 0 flows (1 buckets) sched 65538 weight 0 lmax 0 pri 0 droptail
  sched 65538 type FIFO flags 0x0 0 buckets 0 active
#

```

Resending the 10MB file across the T1 configured line shows these results:

```

root@firewall:~ # iperf3 -F A.bin -c 203.0.113.10 -t 10 --length 1460
Connecting to host 203.0.113.10, port 5201
[ 5] local 203.0.113.50 port 35768 connected to 203.0.113.10 port 5201
[ ID] Interval           Transfer     Bitrate      Retr  Cwnd
[ 5]  0.00-1.01    sec   235 KBytes  1.91 Mbits/sec    0   25.6 KBytes
[ 5]  1.01-2.01    sec   173 KBytes  1.41 Mbits/sec    0   25.6 KBytes
[ 5]  2.01-3.01    sec   195 KBytes  1.60 Mbits/sec    0   27.0 KBytes
[ 5]  3.01-4.01    sec   174 KBytes  1.42 Mbits/sec    0   45.0 KBytes
[ 5]  4.01-5.01    sec   182 KBytes  1.50 Mbits/sec    0   62.1 KBytes
[ 5]  5.01-6.01    sec   178 KBytes  1.46 Mbits/sec    0   62.1 KBytes
[ 5]  6.01-7.01    sec   174 KBytes  1.42 Mbits/sec    0   62.1 KBytes
[ 5]  7.01-8.01    sec   180 KBytes  1.47 Mbits/sec    0   62.1 KBytes
[ 5]  8.01-9.01    sec   204 KBytes  1.67 Mbits/sec    0   62.1 KBytes
[ 5]  9.01-10.01   sec   178 KBytes  1.46 Mbits/sec    0   62.1 KBytes
-----
[ ID] Interval           Transfer     Bitrate      Retr
[ 5]  0.00-10.01    sec   1.83 MBytes  1.53 Mbits/sec    0      sender
      Sent 1.83 MByte / 9.54 MByte (19%) of A.bin
[ 5]  0.00-10.21    sec   1.81 MBytes  1.48 Mbits/sec      receiver

iperf Done.

```

About half of the 3Mbits/sec speed of **pipe 2**, again as expected.

By definition, a **pipe** has just one **queue**, and it is subject to "First In First Out" (FIFO) operation. All traffic that flows through this **pipe** shares the same characteristics.

However, creating a **pipe** also does something else. It creates a default **sched** (scheduler) that governs the **pipe**:

```

Start with no pipes or schedulers

#
# ipfw pipe list
#
# ipfw sched list
#

Create a simple pipe.

# ipfw pipe 1 config bw 100KBit/s
#
# ipfw pipe list
00001: 100.000 Kbit/s    0 ms burst 0
q131073 50 sl. 0 flows (1 buckets) sched 65537 weight 0 lmax 0 pri 0 droptail
      sched 65537 type FIFO flags 0x0 0 buckets 0 active
#

Observe the default scheduler for this pipe

```

```
# ipfw sched list
00001: 100.000 Kbit/s    0 ms burst 0
  sched 1 type WF2Q+ flags 0x0 0 buckets 0 active
#
```

The default scheduler for a new **pipe** is of type **WF2Q+**, a version of the Weighted Fair Queueing algorithm for packet transfer.

This is now a single **pipe** of type FIFO operation that is managed by a **WF2Q+** scheduling algorithm.

The [ipfw\(8\)](#) man page makes note of several other scheduling algorithms. These can be selected by using the "type" keyword on the **pipe** command. The **type** keyword selects the type of scheduler applied to the **pipe** - not the type of the **pipe** itself (the **pipe** remains FIFO):

```
# ipfw pipe list
#
# ipfw sched list
#

Create a pipe and assign a scheduler of type Round Robin (Deficit Round Robin)

# ipfw pipe 1 config bw 100KBit/s type rr
#
# ipfw pipe list
00001: 100.000 Kbit/s    0 ms burst 0
q131073 50 sL. 0 flows (1 buckets) sched 65537 weight 0 lmax 0 pri 0 droptail
  sched 65537 type FIFO flags 0x0 0 buckets 0 active
#

View the new sheduler of type RR (Deficit Round Robin)

# ipfw sched list
00001: 100.000 Kbit/s    0 ms burst 0
  sched 1 type RR flags 0x0 0 buckets 0 active
#
```

*pipes and *sched*s (schedulers) are tightly bound. In fact, there is no command to delete a scheduler. The scheduler is deleted when the pipe is deleted.

Note however that the scheduler can be configured independently if desired. Here is a change to the scheduler type from the above type RR to QFQ, a variant of WF2Q+:

```
#
# ipfw sched 1 config type qfq
```

```

Bump qfq weight to 1 (was 0)
Bump qfq maxlen to 1500 (was 0)
#
# ipfw sched list
00001: 100.000 Kbit/s    0 ms burst 0
  sched 1 type QFQ flags 0x0 0 buckets 0 active
#

```

There are other keywords that can be added to a **pipe** specification: **delay**, **burst**, **profile**, **weight**, **buckets**, **mask**, **noerror**, **plr**, **queue**, **red** or **gred**, **codel**, and **pie**. These are described in the [ipfw\(8\)](#) man page.

A contrived example might be:

```

Start fresh

# ipfw pipe 1 delete
#
# ipfw pipe 1 config bw 100kbit/s delay 20 burst 2000 weight 40 buckets 256 mask src-
ip 0x000000ff noerror plr 0.01 queue 75 red .3/25/30/.5 type qfq
#
# ipfw pipe list
00001: 100.000 Kbit/s    20 ms burst 2000
q131073 75 sl.plr 0.010000 0 flows (1 buckets) sched 65537 weight 40 lmax 0 pri 0
  RED w_q 0.299988 min_th 25 max_th 30 max_p 0.500000
  sched 65537 type FIFO flags 0x1 256 buckets 0 active
  mask: 0x00 0x000000ff/0x0000 -> 0x00000000/0x0000
#
# ipfw sched list
00001: 100.000 Kbit/s    20 ms burst 2000
  sched 1 type QFQ flags 0x1 256 buckets 0 active
  mask: 0x00 0x000000ff/0x0000 -> 0x00000000/0x0000
#

```

Setting up two separate **pipes** to send data to the same destination is overkill. It is like setting up two separate network links between the two points. While that may be desirable for redundancy or high-availability, it makes no difference for bandwidth allocation. (Yes, link aggregation is possible, but that is not being considered here.)

What is usually needed is a way to separate traffic into different "*lanes*" and assign different "*speed limits*" to each lane. That is exactly what **queues** are for.

4.2.2. Simple Pipe and Queue Configuration

Before going further, it is useful to disambiguate the two meanings of the word "**queue**".

In a **pipe** definition, by default, the **pipe** is assigned a **queue** where incoming packets are held

before processing and transit. The size of this "pipe queue" is by default 50 packets, but can be changed with the **queue** keyword on the **pipe** definition:

```
# ipfw pipe 1 config bw 200Kbit/s
#
# ipfw pipe list
00001: 200.000 Kbit/s    0 ms burst 0
q131073 50 sl. 0 flows (1 buckets) sched 65537 weight 0 lmax 0 pri 0 droptail
  sched 65537 type FIFO flags 0x0 0 buckets 0 active
#
# ipfw pipe 2 config bw 200Kbit/s queue 75
#
# ipfw pipe list
00001: 200.000 Kbit/s    0 ms burst 0
q131073 50 sl. 0 flows (1 buckets) sched 65537 weight 0 lmax 0 pri 0 droptail
  sched 65537 type FIFO flags 0x0 0 buckets 0 active
00002: 200.000 Kbit/s    0 ms burst 0
q131074 75 sl. 0 flows (1 buckets) sched 65538 weight 0 lmax 0 pri 0 droptail
  sched 65538 type FIFO flags 0x0 0 buckets 0 active
#
```

In contrast, **dummynet** has the concept of **flow queues** which are virtual groupings of packets assigned to a flow according to a mask in their definition with **ipfw queue** statements.

Configuring a **queue** is almost as simple as configuring a **pipe**.

Start with a clean slate (all objects and rules deleted):

```
# kldunload dummynet
# kldunload ipfw
# kldload ipfw
ipfw2 (+ipv6) initialized, divert loadable, nat loadable, default to deny, logging
disabled
# kldload dummynet
load_dn_sched dn_sched FIFO loaded
load_dn_sched dn_sched QFQ loaded
load_dn_sched dn_sched RR loaded
load_dn_sched dn_sched WF2Q+ loaded
load_dn_sched dn_sched PRIO loaded
load_dn_sched dn_sched FQ_CODEL loaded
load_dn_sched dn_sched FQ_PIE loaded
load_dn_aqm dn_aqm CODEL loaded
load_dn_aqm dn_aqm PIE loaded
#
# ipfw queue 1 config pipe 1
#
# ipfw queue show
```

```
q00001 50 sl. 0 flows (1 buckets) sched 1 weight 0 lmax 0 pri 0 droptail
```

Here is one **queue** of size 50 packets that was created and assigned to **pipe 1**. Since there is no assigned **weight**, the **weight** is 0 (zero), which is the least weight possible. The **queue** currently has 0 flows, meaning that this **queue** has no traffic flowing through it.

Notice however, that the **queue** was created *before* the **pipe**. That is why the weight is 0. The default queue weight is 1. This configuration was actually done out of order. To maintain a readable configuration, it is best to configure the objects in the following order:

1. **pipes** (also creates a scheduler, which can be assigned a specific scheduler type)
2. **queues** - create queues and assign weights, source and destination masks, delay, and other characteristics to the queue
3. Assign **rules** to match traffic using standard 5-tuples or as needed

dummysnet also has the ability to separate out different flows within the same pipe to perform different scheduling algorithms. An example of this capability is shown later in this section.

When transferring a file to the **external1** VM and attempting to type interactively on the **external1** VM at the same time, the ability to type at speed is dramatically reduced. The file transfer packets, being much larger than interactive typing packets are hogging all the bandwidth. This effect is a well known phenomenon to anyone who edits documents on a remote site. Since packets are created much faster by a file transfer program than anyone can type, the outbound queue is almost always full of large packets, leaving keystrokes to be separated by large amounts of file transfer data in the queue.



Try this out on the **firewall** VM by resetting the **pipe 1** bandwidth to 300Kbit/sec, and in one session, run iperf3 as **iperf3 -c 203.0.113.10 -t 60**. Then in another session, add rules for ssh traffic and ssh to **external1** VM and try to enter text into a scratch file. The typing delay is almost unbearable.

To control traffic flow between the **firewall** VM and any external VM host, set up individual **queues** to separate traffic within a **pipe**. **queues** can be either static - defined with **ipfw queue config ...** - or they can be dynamic. Dynamic queues are created when using the **mask** keyword. Masks for **queues** are called **flow masks**. The mask determines if a packet entering or leaving the **firewall** is selected to be entered into a **queue**. Consider the following example:

```
# ipfw pipe 1 config bw 200Kbit/s mask src-ip 0x000000ff
```

Each /24 host transferring data through **pipe 1** (based on suitable rules) will have its own dynamic queue, all sharing the bandwidth in the **pipe** according to the configuration of the queue.

If a different data transfer that is not related to the pipe, queue, and flow mask is started, it will not

have any effect on the data in the pipe and queue. Dummynet keeps such transfers separate from the pipe and queue operations.

If instead, the goal is to create separate individual **queues** with different characteristics such as different **weights** or **delay**, create static **queues** and then assign them to individual **pipes** as desired:

```
#
# ipfw pipe 1 config bw 300kbit/s
#
# ipfw pipe show
00001: 300.000 Kbit/s    0 ms burst 0
q131073 50 sl. 0 flows (1 buckets) sched 65537 weight 0 lmax 0 pri 0 droptail
  sched 65537 type FIFO flags 0x0 0 buckets 0 active
#
# ipfw queue 1 config pipe 1 weight 10 mask dst-ip 0xffffffff dst-port 5201
Bump flowset buckets to 64 (was 0)
#
# ipfw queue 2 config pipe 1 weight 10 mask dst-ip 0xffffffff dst-port 5202
Bump flowset buckets to 64 (was 0)
#
# ipfw queue show
q00001 50 sl. 0 flows (64 buckets) sched 1 weight 10 lmax 0 pri 0 droptail
  mask: 0x00 0x00000000/0x0000 -> 0xffffffff/0x1451
q00002 50 sl. 0 flows (64 buckets) sched 1 weight 10 lmax 0 pri 0 droptail
  mask: 0x00 0x00000000/0x0000 -> 0xffffffff/0x1452
#
# ipfw add 10 allow icmp from any to any
00010 allow icmp from any to any
#
# ipfw add 100 check-state
00100 check-state :default
#
# ipfw add 1000 queue 1 tcp from me to 203.0.113.10 5201 setup keep-state
01000 queue 1 tcp from me to 203.0.113.10 5201 setup keep-state :default
#
# ipfw add 1100 queue 2 tcp from me to 203.0.113.20 5202 setup keep-state
01100 queue 2 tcp from me to 203.0.113.20 5202 setup keep-state :default
#
# ipfw list
00010 allow icmp from any to any
00100 check-state :default
01000 queue 1 tcp from me to 203.0.113.10 5201 setup keep-state :default
01100 queue 2 tcp from me to 203.0.113.20 5202 setup keep-state :default
65535 deny ip from any to any
#
```



Later versions of FreeBSD may not return any output on **ipfw queue** configuration statements. The configuration is completed successfully, though without any

output.

Running

```
# iperf3 -c 203.0.113.10 -p 5201 -t 30 -O 5 --length 1460
```

produces the output below.

The output is the result of using the "omit" flag (-O) on the sender to ignore the first five seconds of output. This removes the "slow start" portion of the TCP test, and focuses instead on the "steady state" that occurs after slow start gets up to speed.

```
Sender: firewall, Receiver: external1]
# iperf3 -c 203.0.113.10 -p 5201 -t 30 -O 5 --length 1460
Connecting to host 203.0.113.10, port 5201
[ ID] Interval           Transfer             Bitrate
[  6]  0.00-1.01         sec  44.2 KBytes        359 Kbits/sec
[  6]  1.01-2.01         sec  29.9 KBytes        245 Kbits/sec
[  6]  2.01-3.01         sec  38.5 KBytes        315 Kbits/sec
[  6]  3.01-4.01         sec  35.6 KBytes        292 Kbits/sec
[  6]  4.01-5.01         sec  37.1 KBytes        304 Kbits/sec
[  6]  5.01-6.02         sec  34.2 KBytes        277 Kbits/sec
[  6]  6.02-7.01         sec  32.8 KBytes        271 Kbits/sec
[  6]  7.01-8.02         sec  34.2 KBytes        277 Kbits/sec
[  6]  8.02-9.01         sec  38.5 KBytes        319 Kbits/sec
[  6]  9.01-10.02        sec  38.5 KBytes        312 Kbits/sec
[  6] 10.02-11.01        sec  32.8 KBytes        271 Kbits/sec
[  6] 11.01-12.01        sec  34.2 KBytes        280 Kbits/sec
[  6] 12.01-13.01        sec  41.3 KBytes        339 Kbits/sec
[  6] 13.01-14.02        sec  31.4 KBytes        254 Kbits/sec
[  6] 14.02-15.01        sec  34.2 KBytes        283 Kbits/sec
[  6] 15.01-16.01        sec  38.5 KBytes        315 Kbits/sec
[  6] 16.01-17.01        sec  28.5 KBytes        234 Kbits/sec
[  6] 17.01-18.01        sec  41.3 KBytes        339 Kbits/sec
[  6] 18.01-19.02        sec  32.8 KBytes        266 Kbits/sec
[  6] 19.02-20.01        sec  32.8 KBytes        271 Kbits/sec
[  6] 20.01-21.01        sec  44.2 KBytes        362 Kbits/sec
[  6] 21.01-22.02        sec  31.4 KBytes        254 Kbits/sec
[  6] 22.02-23.01        sec  34.2 KBytes        283 Kbits/sec
[  6] 23.01-24.01        sec  34.2 KBytes        280 Kbits/sec
[  6] 24.01-25.02        sec  42.8 KBytes        347 Kbits/sec
[  6] 25.02-26.01        sec  32.8 KBytes        271 Kbits/sec
[  6] 26.01-27.01        sec  32.8 KBytes        269 Kbits/sec
[  6] 27.01-28.01        sec  32.8 KBytes        269 Kbits/sec
[  6] 28.01-29.01        sec  41.3 KBytes        339 Kbits/sec
[  6] 29.01-30.00        sec  34.2 KBytes        283 Kbits/sec
- - - - -
[ ID] Interval           Transfer             Bitrate
[  6]  0.00-30.00        sec  1.05 MBytes        293 Kbits/sec sender
[  6]  0.00-31.52        sec  1.10 MBytes        292 Kbits/sec receiver
```

Figure 20. Testing Separate Static Queues and Pipes

This example shows the steady state results of transmitting data through one queue - **queue 1**. The bitrate was consistently about 293Kbits/sec.



Later versions of FreeBSD and **iperf3** may differ from the display in the above figure. Assess the correctness of the queue setup by examining the transfer summary printed at the end of the **iperf3** command output. Use of the **iperf3 --length** parameter may provide additional clarity for transfers.

During the transmission, a view of the queue status was:

```
# ipfw queue show
q00001 50 sl. 2 flows (64 buckets) sched 1 weight 10 lmax 0 pri 0 droptail
mask: 0x00 0x00000000/0x0000 -> 0xffffffff/0x1451
BKT Prot Source IP/port Dest. IP/port Tot_pkt/bytes Pkt/Byte Drp
136 ip 0.0.0.0/0 203.0.113.10/5201 2293 3425216 42 63000 0
50 ip 0.0.0.0/0 203.0.113.50/1040 752 39104 1 52 0
q00002 50 sl. 0 flows (64 buckets) sched 1 weight 10 lmax 0 pri 0 droptail
mask: 0x00 0x00000000/0x0000 -> 0xffffffff/0x1452
#
```

The **queue mask**, set to show the full destination address and destination port is highlighted.



Note that port numbers are displayed in hexadecimal. A decimal/hexadecimal calculator may be helpful when looking at a lot of queue displays.

The next example shows the result of starting two transmissions, one for each queue.

On the **external1** VM, set up the command **iperf3 -s -p 5201**, and on **external2** use the command **iperf3 -s -p 5202**.

Start the transfer to **external1** on the **firewall** VM with the command:

```
# iperf3 -c 203.0.113.10 -p 5201 -t 180 -O 30
```

and start the second transfer from a different session on the **firewall** VM with the command:

```
# iperf3 -c 203.0.113.20 -p 5202 -t 180 -O 30
```

Notice how the queue is adjusted to accommodate the presence of a second queue of equal weight:

Sender1 output (firewall) through queue 1 weight 10				Sender2 output (firewall) through queue 2 weight 10			
# iperf3 -c 203.0.113.10 -p 5201 -t 50 -o 5 --length 1460				# iperf3 -c 203.0.113.20 -p 5202 -t 50 -o 5 --length 1460			
Connecting to host 203.0.113.10, port 5201				Connecting to host 203.0.113.20, port 5202			
[6] connected to 203.0.113.10 port 5201				[6] connected to 203.0.113.20 port 5202			
[ID]	Interval	Transfer	Bitrate	[ID]	Interval	Transfer	Bitrate
[6]	0.00-1.01	sec 37.1 KBytes	301 Kbits/sec	[6]	0.00-1.01	sec 17.1 KBytes	139 Kbits/sec
[6]	1.01-2.01	sec 35.6 KBytes	292 Kbits/sec	[6]	1.01-2.02	sec 18.5 KBytes	150 Kbits/sec
[6]	2.01-3.02	sec 32.8 KBytes	266 Kbits/sec	[6]	2.02-3.02	sec 15.7 KBytes	128 Kbits/sec
[6]	3.02-4.01	sec 37.1 KBytes	307 Kbits/sec	[6]	3.02-4.02	sec 22.8 KBytes	187 Kbits/sec
[6]	4.01-5.01	sec 35.6 KBytes	292 Kbits/sec	[6]	4.02-5.02	sec 14.3 KBytes	117 Kbits/sec
[6]	5.01-6.01	sec 34.2 KBytes	280 Kbits/sec	[6]	5.02-6.01	sec 18.5 KBytes	153 Kbits/sec
[6]	6.01-7.02	sec 35.6 KBytes	289 Kbits/sec	[6]	6.01-7.02	sec 17.1 KBytes	139 Kbits/sec
[6]	7.02-8.01	sec 31.4 KBytes	260 Kbits/sec	[6]	7.02-8.02	sec 17.1 KBytes	140 Kbits/sec
[6]	8.01-9.02	sec 38.5 KBytes	312 Kbits/sec	[6]	8.02-9.01	sec 18.5 KBytes	153 Kbits/sec
[6]	9.02-10.01	sec 34.2 KBytes	283 Kbits/sec	[6]	9.01-10.02	sec 20.0 KBytes	162 Kbits/sec
[6]	10.01-11.02	sec 37.1 KBytes	301 Kbits/sec	[6]	10.02-11.01	sec 15.7 KBytes	130 Kbits/sec
[6]	11.02-12.01	sec 34.2 KBytes	283 Kbits/sec	[6]	11.01-12.02	sec 15.7 KBytes	127 Kbits/sec
[6]	12.01-13.01	sec 39.9 KBytes	327 Kbits/sec	[6]	12.02-13.02	sec 20.0 KBytes	163 Kbits/sec
[6]	13.01-14.02	sec 32.8 KBytes	266 Kbits/sec	[6]	13.02-14.02	sec 18.5 KBytes	152 Kbits/sec
[6]	14.02-15.01	sec 34.2 KBytes	283 Kbits/sec	[6]	14.02-15.02	sec 18.5 KBytes	152 Kbits/sec
[6]	15.01-16.01	sec 35.6 KBytes	292 Kbits/sec	[6]	15.02-16.02	sec 18.5 KBytes	152 Kbits/sec
[6]	16.01-17.02	sec 31.4 KBytes	254 Kbits/sec	[6]	16.02-17.02	sec 15.7 KBytes	128 Kbits/sec
[6]	17.02-18.01	sec 39.9 KBytes	330 Kbits/sec	[6]	17.02-18.02	sec 17.1 KBytes	140 Kbits/sec
[6]	18.01-19.02	sec 32.8 KBytes	266 Kbits/sec				
[6]	19.02-20.01	sec 31.4 KBytes	260 Kbits/sec				
[6]	20.01-21.01	sec 15.7 KBytes	128 Kbits/sec				
[6]	21.01-22.01	sec 20.0 KBytes	164 Kbits/sec				
[6]	22.01-23.01	sec 14.3 KBytes	117 Kbits/sec				
[6]	23.01-24.01	sec 17.1 KBytes	140 Kbits/sec				
[6]	24.01-25.01	sec 21.4 KBytes	175 Kbits/sec				
[6]	25.01-26.01	sec 15.7 KBytes	128 Kbits/sec				
[6]	26.01-27.01	sec 15.7 KBytes	128 Kbits/sec				
[6]	27.01-28.01	sec 18.5 KBytes	152 Kbits/sec				
[6]	28.01-29.01	sec 15.7 KBytes	128 Kbits/sec				
[6]	29.01-30.01	sec 22.8 KBytes	187 Kbits/sec				
[6]	30.01-31.01	sec 14.3 KBytes	117 Kbits/sec				
[6]	31.01-32.01	sec 20.0 KBytes	164 Kbits/sec				
[6]	32.01-33.01	sec 15.7 KBytes	128 Kbits/sec				
[6]	33.01-34.01	sec 17.1 KBytes	140 Kbits/sec				
[6]	34.01-35.01	sec 21.4 KBytes	175 Kbits/sec				
[6]	35.01-36.02	sec 14.3 KBytes	116 Kbits/sec				
[6]	36.02-37.01	sec 20.0 KBytes	165 Kbits/sec				
[6]	37.01-38.01	sec 17.1 KBytes	140 Kbits/sec				

Figure 21. Testing Two Static Queues and Pipes

Since the queues were equally weighted, the result was that the transmission bitrate for both queues was reduced to about half of the transmission bitrate before the second transmission started.

The highlighted area shows how the first queue adapted.

Queue characteristics can be changed at any time, even during an active flow. Consider the case below where, during simultaneous transmission through queues of equal weight, the queue weight of the second queue was modified as follows:

queue 1: original weight 10 modified weight 10 (no change)

queue 2: original weight 10 modified weight 50 (increased)

This change can be effected by the command:

```
# ipfw queue 2 config weight 50
```

Sender1 output (firewall) through queue 1 weight 10					Sender2 output (firewall) through queue 2 weight 10				
# iperf3 -c 203.0.113.10 -p 5201 -t 90 -o 5 --length 1460					# iperf3 -c 203.0.113.20 -p 5202 -t 90 -o 5 --length 1460				
[6]	43.01-44.01	sec	17.1 KBytes	140 Kbits/sec
[6]	44.01-45.01	sec	18.5 KBytes	152 Kbits/sec
[6]	45.01-46.01	sec	14.3 KBytes	117 Kbits/sec
[6]	46.01-47.01	sec	20.0 KBytes	164 Kbits/sec
[6]	47.01-48.02	sec	17.1 KBytes	139 Kbits/sec
[6]	48.02-49.02	sec	18.5 KBytes	152 Kbits/sec
[6]	49.02-50.01	sec	15.7 KBytes	130 Kbits/sec
[6]	50.01-51.01	sec	21.4 KBytes	175 Kbits/sec
[6]	51.01-52.01	sec	14.3 KBytes	117 Kbits/sec
[6]	52.01-53.02	sec	18.5 KBytes	150 Kbits/sec
[6]	53.02-54.01	sec	17.1 KBytes	142 Kbits/sec
[6]	54.01-55.01	sec	14.3 KBytes	117 Kbits/sec
[6]	55.01-56.01	sec	7.13 KBytes	58.4 Kbits/sec	# ipfw queue 2 config weight 50
[6]	56.01-57.01	sec	5.70 KBytes	46.7 Kbits/sec
[6]	57.01-58.01	sec	2.85 KBytes	23.4 Kbits/sec
[6]	58.01-59.01	sec	5.70 KBytes	46.7 Kbits/sec
[6]	59.01-60.01	sec	7.13 KBytes	58.4 Kbits/sec
[6]	60.01-61.01	sec	2.85 KBytes	23.4 Kbits/sec
[6]	61.01-62.01	sec	8.55 KBytes	70.1 Kbits/sec
[6]	62.01-63.01	sec	5.70 KBytes	46.7 Kbits/sec
[6]	63.01-64.01	sec	5.70 KBytes	46.7 Kbits/sec
[6]	64.01-65.01	sec	5.70 KBytes	46.8 Kbits/sec
[6]	65.01-66.01	sec	7.13 KBytes	58.4 Kbits/sec
[6]	66.01-67.01	sec	5.70 KBytes	46.7 Kbits/sec
[6]	67.01-68.01	sec	5.70 KBytes	46.7 Kbits/sec
[6]	68.01-69.01	sec	7.13 KBytes	58.4 Kbits/sec

Figure 22. Testing Two Static Queues and Pipes Changed In-flight

The transmission bitrate for **queue 1** dropped from an average of about 140Kbits/sec to an average of about 50Kbits/sec; while the rate for **queue 2** expanded during and after the reconfiguration.

Note however, that the above command had a side effect:

```
# ipfw queue show
q00001 50 sl. 0 flows (64 buckets) sched 1 weight 10 lmax 0 pri 0 droptail
    mask: 0x00 0x00000000/0x0000 -> 0xffffffff/0x1451
q00002 50 sl. 0 flows (1 buckets) sched 1 weight 50 lmax 0 pri 0 droptail
#
```

The **flow mask** for **queue 2** has been deleted. In fact, **all** settings not explicitly reset will revert to their default settings. Here is a complicated queue setup:

```
# ipfw queue 1 config pipe 1 weight 40 buckets 256 mask src-ip 0x000000ff dst-ip
0x0000ffff noerror plr 0.01 queue 75 red .3/25/30/.5
#
# ipfw queue show
q00001 75 sl.plr 0.010000 0 flows (256 buckets) sched 1 weight 40 lmax 0 pri 0
    RED w_q 0.299988 min_th 25 max_th 30 max_p 0.500000
    mask: 0x00 0x000000ff/0x0000 -> 0x0000ffff/0x0000
#
```

And if, similar to the previous example, only the weight is changed:

```
# ipfw queue 1 config weight 20
```

```
#  
# ipfw queue show  
q00001 50 sl. 0 flows (1 buckets) sched 1 weight 20 lmax 0 pri 0 droptail  
#
```

All the other parameters of the queue are reset to their defaults. Therefore, *it is best to retain the original commands used to construct queues, pipes, and schedulers, even if changing only one parameter*. That way, all other parameters can be replicated on the command line. Otherwise it may be necessary to reconstruct the parameters from the output of **ipfw queue show** which can be quite tedious.

4.2.3. Relationships

As described throughout this section, **pipes**, **queues**, and **scheds** (schedulers) are interrelated. Here are some simplified principles:

- **Bandwidth** - the bandwidth of a particular pipe determines the highest rate at which all data will move through the pipe with optimal conditions. With lower configured bandwidth, less data will be transferred. This has an effect on **queue** size.
- **Queue size** - the number of packets, or if expressed in K or Mbytes, the amount of data waiting to be transferred through a pipe. If the queue fills up or overflows, packets are dropped which may result in retransmissions, depending on the protocol or application involved. That being said, best practice is to configure for smaller, rather than larger queue sizes. See [RFC 2309](#) for a thorough discussion.
- **Delay** - delay can be configured in a pipe to inject additional time between individual transfers. It is distinct from bandwidth in that it can only slow down traffic, not speed it up.
- **Packet Loss** - packet loss can be configured in a pipe to simulate lossy transmission media. It simulates how well the receiver can correctly "hear" the transmissions. Packet loss may also result in retransmissions.
- **Scheduling** - scheduling determines the allocation of bandwidth among flows. If there is only one queue in a pipe, and one flow in that queue, the scheduler does not really have much to do. However, if there are multiple queues in a pipe each with their own flow, the scheduler determines the order of service based on the selected algorithm (RR, QFQ, WFQ+, etc.) and queue **weights**.
- **Queue weight** - a numerical value that can be used to influence the scheduler to prefer certain flows ahead of other flows. Higher weights result in increased traffic in a flow. However, even with a very minimal weight, a flow will never starve - that is, it will still eventually get serviced by the scheduler.

Additional detail is contained in [ipfw\(8\)](#).

4.2.4. Dynamic Pipes

Here, note that the simplest setup for **pipes** creates **dynamic pipes** when needed:

```
# ipfw pipe 1 config bw 300kbit/s weight 10 mask src-ip 0x0000ffff dst-ip 0xffffffff
```

```

Bump sched buckets to 64 (was 0)
#
# ipfw pipe show
00001: 300.000 Kbit/s    0 ms burst 0
q131073 50 sL. 0 flows (1 buckets) sched 65537 weight 10 lmax 0 pri 0 droptail
  sched 65537 type FIFO flags 0x1 64 buckets 0 active
    mask: 0x00 0x0000ffff/0x0000 -> 0xffffffff/0x0000
#
# ipfw list
00050 allow icmp from any to any
00100 check-state :default
65535 deny ip from any to any
#
# ipfw add 1000 pipe 1 tcp from me to 203.0.113.0/24 5201-5203 setup keep-state
01000 pipe 1 tcp from me to 203.0.113.0/24 5201-5203 setup keep-state :default
#
# ipfw list
01000 pipe 1 tcp from me to 203.0.113.0/24 5201-5203 setup keep-state :default
65535 deny ip from any to any
#

```

Sending some data with this configuration:

```

# ipfw pipe show
00001: 300.000 Kbit/s    0 ms burst 0
q131073 50 sL. 0 flows (1 buckets) sched 65537 weight 10 lmax 0 pri 0 droptail
  sched 65537 type FIFO flags 0x1 64 buckets 4 active
    mask: 0x00 0x0000ffff/0x0000 -> 0xffffffff/0x0000
BKT Prot    Source IP/port      Dest. IP/port      Tot_pkt/bytes Pkt/Byte Drp
  6 ip       0.0.10.10/0        203.0.113.50/0    236   12272 0   0 0
 78 ip       0.0.10.50/0        203.0.113.10/0   1493  2225216 43 64500 0
 80 ip       0.0.10.50/0        203.0.113.20/0   1355  2018216 42 63000 0
 58 ip       0.0.10.20/0        203.0.113.50/0    366   19032 0   0 0
#
# ipfw list
00050 allow icmp from any to any
00100 check-state :default
01000 pipe 1 tcp from me to 203.0.113.0/24 5201-5203 setup keep-state :default
65535 deny ip from any to any
#

```

All three transmissions running together, single **pipe**:

```

# ipfw pipe show
00001: 300.000 Kbit/s    0 ms burst 0
q131073 50 sL. 0 flows (1 buckets) sched 65537 weight 10 lmax 0 pri 0 droptail
  sched 65537 type FIFO flags 0x1 64 buckets 6 active
    mask: 0x00 0x0000ffff/0x0000 -> 0xffffffff/0x0000
BKT Prot    Source IP/port      Dest. IP/port      Tot_pkt/bytes Pkt/Byte Drp

```

6 ip	0.0.10.10/0	203.0.113.50/0	588	30576	0	0	0
78 ip	0.0.10.50/0	203.0.113.10/0	1508	2247716	43	64500	0
80 ip	0.0.10.50/0	203.0.113.20/0	1357	2021216	43	64500	0
90 ip	0.0.10.50/0	203.0.113.30/0	1322	1981552	41	61500	0
46 ip	0.0.10.30/0	203.0.113.50/0	34	1768	0	0	0
58 ip	0.0.10.20/0	203.0.113.50/0	702	36504	0	0	0

Because of the **ipfw** rule:

```
01000 pipe 1 tcp from me to 203.0.113.0/24 5201-5203 setup keep-state :default
```

All are getting 290 Kbit/sec from **iperf3** and they are all sharing the **pipe** equally.

If **iperf3** is changed to send to different ports for each system (5201, 5202, 5203) on the **external1**, **external2**, and **external3** VMs respectively, there is no change. It is only with **queues**, and setting the individual flow rate, that can effect change.

Below are examples of different masks and their effect on traffic flow:

```
* dst-ip 0x0000ffff
```

```
# ipfw pipe show
```

```
00001: 300.000 Kbit/s    0 ms burst 0
```

```
q131073 50 sl. 0 flows (1 buckets) sched 65537 weight 0 lmax 0 pri 0 droptail
sched 65537 type FIFO flags 0x1 64 buckets 4 active
```

```
mask: 0x00 0x00000000/0x0000 -> 0x0000ffff/0x0000
```

BKT	Prot	Source IP/port	Dest. IP/port	Tot_pkt/bytes	Pkt/Byte	Drp
10	ip	0.0.0.0/0	0.0.10.10/0	1183 1760218	43 64500	0
20	ip	0.0.0.0/0	0.0.10.20/0	974 1446718	42 63000	0
30	ip	0.0.0.0/0	0.0.10.30/0	688 1017718	35 52500	0
50	ip	0.0.0.0/0	0.0.10.50/0	1717 89284	0 0	0

```
* dst-ip 0xffffffff
```

```
# ipfw pipe show
```

```
00001: 300.000 Kbit/s    0 ms burst 0
```

```
q131073 50 sl. 0 flows (1 buckets) sched 65537 weight 0 lmax 0 pri 0 droptail
sched 65537 type FIFO flags 0x1 64 buckets 4 active
```

```
mask: 0x00 0x00000000/0x0000 -> 0xffffffff/0x0000
```

BKT	Prot	Source IP/port	Dest. IP/port	Tot_pkt/bytes	Pkt/Byte	Drp
18	ip	0.0.0.0/0	203.0.113.50/0	402 20888	0 0	0
42	ip	0.0.0.0/0	203.0.113.10/0	144 204722	0 0	0
52	ip	0.0.0.0/0	203.0.113.20/0	359 525971	0 0	0
62	ip	0.0.0.0/0	203.0.113.30/0	562 843000	37 55500	0

```
* src-ip 0x0000ffff
```


ipfw pipe show

00001: 300.000 Kbit/s 0 ms burst 0

q131073 50 sL. 0 flows (1 buckets) sched 65537 weight 0 lmax 0 pri 0 droptail
sched 65537 type FIFO flags 0x1 64 buckets 4 active

mask: 0x00 0x0000ffff/0x0000 -> 0x00000000/0x0000

BKT	Prot	Source IP/port	Dest. IP/port	Tot_pkt/bytes	Pkt/Byte	Drp
20	ip	0.0.10.10/0	0.0.0.0/0	361	19348 0 0 0	
100	ip	0.0.10.50/0	0.0.0.0/0	2102	3079974 36 54000 27	
40	ip	0.0.10.20/0	0.0.0.0/0	193	10416 0 0 0	
60	ip	0.0.10.30/0	0.0.0.0/0	47	2612 0 0 0	

* mask src-ip 0x0000ffff dst-ip 0x0000ffff <-only one keyword mask needs to be specified

ipfw pipe show

00001: 300.000 Kbit/s 0 ms burst 0

q131073 50 sL. 0 flows (1 buckets) sched 65537 weight 0 lmax 0 pri 0 droptail
sched 65537 type FIFO flags 0x1 64 buckets 6 active

mask: 0x00 0x0000ffff/0x0000 -> 0x0000ffff/0x0000

BKT	Prot	Source IP/port	Dest. IP/port	Tot_pkt/bytes	Pkt/Byte	Drp
14	ip	0.0.10.30/0	0.0.10.50/0	253	13156 0 0 0	
26	ip	0.0.10.20/0	0.0.10.50/0	61	3172 0 0 0	
38	ip	0.0.10.10/0	0.0.10.50/0	771	40094 0 0 0	
110	ip	0.0.10.50/0	0.0.10.10/0	853	1265218 40 60000 0	
112	ip	0.0.10.50/0	0.0.10.20/0	723	1083052 37 55500 0	
122	ip	0.0.10.50/0	0.0.10.30/0	644	951718 34 51000 0	

* mask src-ip 0x0000ffff dst-ip 0x0000ffff dst-port 5201

ipfw pipe show

00001: 300.000 Kbit/s 0 ms burst 0

q131073 50 sL. 0 flows (1 buckets) sched 65537 weight 0 lmax 0 pri 0 droptail
sched 65537 type FIFO flags 0x1 64 buckets 6 active

mask: 0x00 0x0000ffff/0x0000 -> 0x0000ffff/0x1451

BKT	Prot	Source IP/port	Dest. IP/port	ot_pkt/bytes	Pkt/Byte	Drp
204	ip	0.0.10.50/0	0.0.10.10/5201	2132	3183718 43 64500 0	
14	ip	0.0.10.30/0	0.0.10.50/4096	823	42796 0 0 0	
210	ip	0.0.10.50/0	0.0.10.20/5201	2001	2987218 43 64500 0	
152	ip	0.0.10.20/0	0.0.10.50/4161	663	34476 0 0 0	
216	ip	0.0.10.50/0	0.0.10.30/5201	1981	2957218 43 64500 0	
164	ip	0.0.10.10/0	0.0.10.50/65	471	24492 0 0 0	

* mask src-ip 0xffffffff dst-ip 0xffffffff

ipfw pipe 1 show

00001: 300.000 Kbit/s 0 ms burst 0

q131073 50 sL. 0 flows (1 buckets) sched 65537 weight 0 lmax 0 pri 0 droptail
sched 65537 type FIFO flags 0x1 64 buckets 6 active

```
mask: 0x00 0xffffffff/0x0000 -> 0xffffffff/0x0000
```

BKT	Prot	Source IP/port	Dest. IP/port	Tot_pkt/bytes	Pkt/Byte	Drp
64	ip	203.0.113.50/0	203.0.113.20/0	1215 1808218	43 64500	0
74	ip	203.0.113.50/0	203.0.113.30/0	1023 1533052	43 64500	0
22	ip	203.0.113.10/0	203.0.113.50/0	746 38792	0 0	0
94	ip	203.0.113.50/0	203.0.113.10/0	1863 2780218	42 63000	0
42	ip	203.0.113.20/0	203.0.113.50/0	481 25012	0 0	0
62	ip	203.0.113.30/0	203.0.113.50/0	159 8268	0 0	0

4.2.5. Other Pipe and Queue Commands

To delete pipes and queues use the following syntax:

For queues, specify the queue number on the command line:

```
# ipfw queue delete 1
```

For pipes, specify the pipe number on the command line:

```
# ipfw pipe delete 1
```

Note however that:

```
# ipfw delete pipe 1 <----- does not throw error, and does not delete the pipe.
```

The same is true for the corresponding **queue** keyword. Take care to use the proper syntax.



It is possible to delete a **pipe** with a **pipe** statement still in the ruleset. **ipfw** will not throw an error - but any data transfer matching the **pipe** statement will not work.

scheds (schedulers) and **pipes** are tightly bound. To delete a scheduler, first delete the **pipe**, and then re-create the **pipe**. The scheduler for the new **pipe** is reset to the default scheduler. However, it is possible to change the current scheduler type at any time:

To change the scheduler type:

```
# ipfw sched config 1 type wfq2 # or rr or any other sched type
```

4.3. Adding Additional Virtual Machines

Up to this point, only two or three virtual machines have been used for exploring **ipfw**. The later material in this book requires the use of several additional virtual machines.

The NAT chapter calls for several more VMs for:

Static NAT Configuration

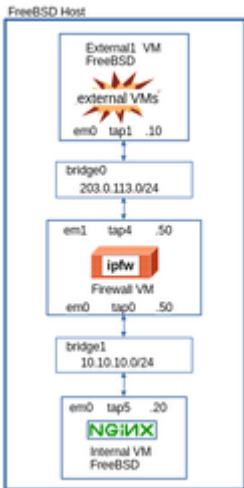


Figure 23. Setting Up Simple NAT

LSNAT Configuration

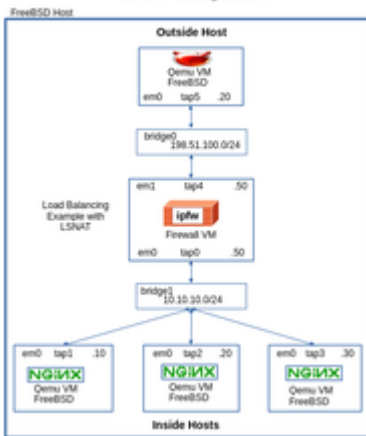


Figure 24. Setting up Load Sharing NAT

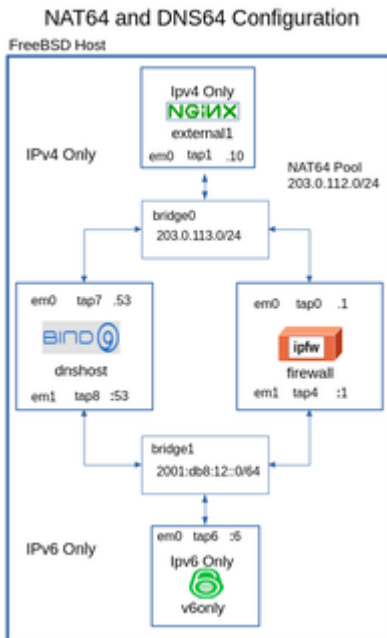


Figure 25. Setting Up NAT64 and DNS64

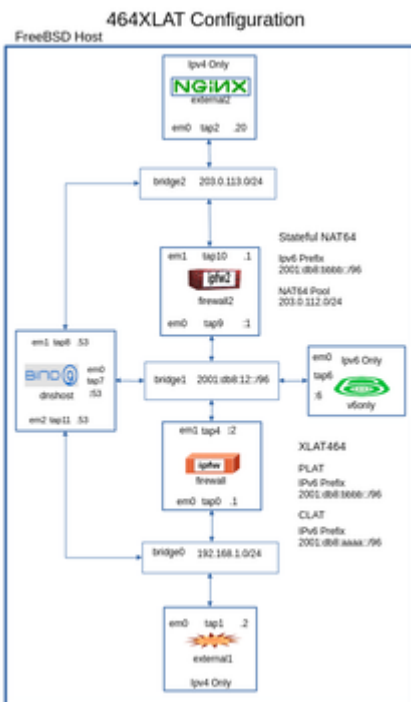


Figure 26. Setting Up 464XLAT

If you have not already done so, finish setting up the remaining VMs as described in [Appendix A](#).

Also, ensure each virtual machine is set up to boot a serial console by adding "console=comconsole" to /boot/loader.conf.

Finally, adjust the number of active windows in **swim.sh** (or **scim.sh**) by uncommenting the appropriate lines in the script.

Chapter 5. ipfw NAT

Network Address Translation (**NAT**) is the process of changing the source or destination address of a packet as it flows through the firewall. This is done chiefly to segregate internal networks and subnets from external networks.

FreeBSD has two capabilities for NAT - `natd(8)` a daemon process that can perform these translations, and in-kernel NAT with `ipfw`. Both of these capabilities use the `libalias(3)` library. This section will focus primarily on in-kernel NAT with `ipfw`.

5.1. General Procedures for Working NAT Examples

This section uses more than two virtual machines (VMs). Begin the setup for simple NAT by following the directions on [Setting Up the Entire IPFW Lab](#).

The examples in this section and later sections grow increasingly complex. Follow this standard procedure for startup with each new example:

1. On the host, set up the bridge and tap setup needed for the examples. Use `mkbr.sh` to configure bridge and tap devices on the host. Examine the figure, and run the script with all bridges and taps accounted for.
2. Start up `swim.sh` (or `scim.sh`) for access to VM serial consoles.
3. Start up the required VMs. Use `runvm.sh` to start up several VMs at one time.
4. On each VM, set up the required addressing. Check the diagram in each Section for addressing requirements.
5. Ensure all VMs have connectivity to their local network peers.
6. If there are additional scripts to load onto the `firewall`, `external`, `internal`, `dnshost`, or `v6only` VMs, load them.
7. If there are specific DNS entries that are required for an example, load them into the `dnshost` and test the entries from another VM.
8. Other VMs in some examples require adding additional routes.
9. On the `firewall` VM, unload and reload the firewall: (`kldunload ipfw` and `kldload ipfw`).
10. Check whether any `sysctl` entries are required for the example.
11. Follow the procedure given for each section.
12. Troubleshoot as necessary.

5.2. Setting Up for Simple NAT

Static NAT Configuration

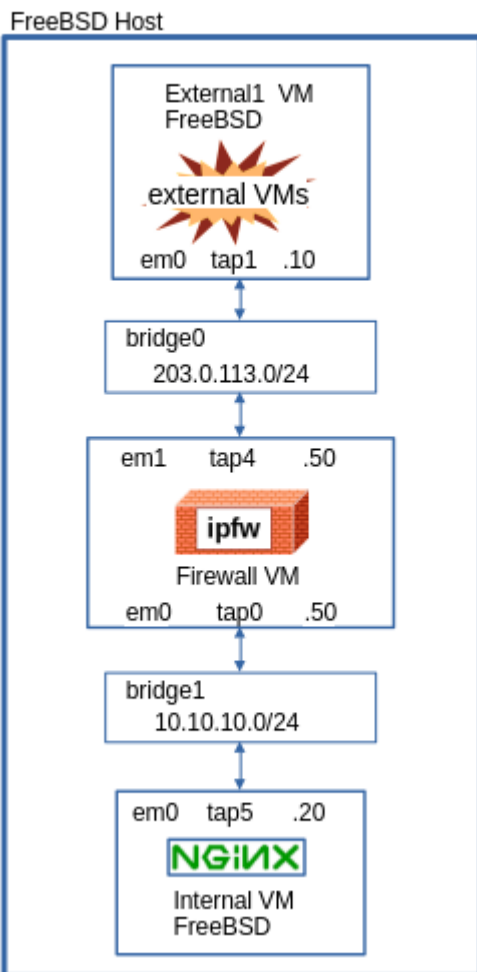


Figure 27. Setting Up Simple NAT

Shut down the existing VMs from the previous examples and reload **ipfw**. To set up the correct network bridge and tap architecture as shown in the figure above, use this command:

```
# sudo /bin/sh mkbr.sh reset bridge0 tap1 tap4 bridge1 tap0 tap5
```

Restart the desired VMs with:

```
# /bin/sh runvm.sh firewall external1 internal
```

Use the above figure to set up the correct addresses for each VM and ping adjacent interfaces.

For routing, the **external1** VM should have the default route set to 203.0.113.50. The **internal** VM should have its default route set to 10.10.10.50. The **firewall** should have its default route set to 203.0.113.10 (**external1**) since this example wants all traffic to exit via the **firewall** **em1** interface.

The **firewall** should already be set up for IP forwarding (`sysctl net.inet.ip.forwarding=1`), but if

not, set the sysctl as indicated. Try to ping em0 on **external1** VM from the **internal** VM host and vice-versa. Check all addressing, the host bridge and tap devices, and the sysctl `net.inet.ip.forwarding=1` on the firewall if something is not working.

On the **firewall** VM, restart **ipfw** with

```
# kldload ipfw
```

To use in-kernel NAT, first load the **ipfw_nat** kernel module:

```
# kldload ipfw_nat
```

Running **kldstat** should now show output similar to:

```
# kldstat
Id Refs Address          Size Name
 1   11 0xffffffff80200000 1f370e8 kernel
 2    1 0xffffffff82818000   3220 intpm.ko
 3    1 0xffffffff8281c000   2178 smb.ko
 4    2 0xffffffff8281f000  27450 ipfw.ko
 5    1 0xffffffff82847000   42d0 ipfw_nat.ko
 6    1 0xffffffff8284c000   c962 libalias.ko
#
```

The example is ready to explore **ipfw_nat**.

Similar to other **ipfw** entities such as *pipes* and *queues*, **ipfw_nat** works with a NAT *object*. A NAT object is a single entry in the packet aliasing database.

First, create a NAT object:

```
# ipfw nat 25 config ip 198.51.100.50
ipfw nat 25 config ip 198.51.100.50
#
# ipfw nat show config
ipfw nat 25 config ip 198.51.100.50
```

Note that the NAT object identifier *must be numeric*, not alphabetic or alphanumeric. A NAT object identifier such as `foo` or `25foo` will be rejected by **ipfw**.

Next, load two rules that will use that instance:

```
# ipfw add 1000 nat 25 tcp from any to any
# ipfw add 2000 nat 25 icmp from any to any
```

Listing the ruleset shows the NAT object and the rule body.

```
# ipfw list
01000 nat 25 tcp from any to any
02000 nat 25 icmp from any to any
65535 deny ip from any to any
```

There is now an **ipfw_nat** instance in the packet aliasing database and rules that will engage that instance. This is generally referred to as "static NAT".

The **ipfw_nat** instance will replace the IP source address of any packet exiting the firewall with 198.51.100.50, provided that packet has reached the **ipfw_nat** rule and matches its configuration.

To test, start [tcpdump\(1\)](#) on the host system monitoring **bridge0**. (Ensure once again that the host system is not running a firewall.)

```
host_system# tcpdump -n -i bridge0 -v
```

Then, from the **firewall** VM, [telnet\(1\)](#) to any IP address **not** used in the lab:

```
# telnet 172.16.10.10
Trying 172.16.10.10...
^C
```

A few seconds to attempt the connection (which will not succeed anyway) shows the host **tcpdump** output:

```
host_system# tcpdump -n -i bridge0 -v
tcpdump: listening on bridge0, link-type EN10MB (Ethernet), snapshot length 262144
bytes
19:58:34.099782 IP (tos 0x10, ttl 64, id 0, offset 0, flags [DF], proto TCP (6),
length 60)
    198.51.100.50.62143 > 172.16.10.10.23: Flags [S], cksum 0x89d4 (correct), seq
3107170690, win 65535, options [mss 1460,nop,wscale 6,sackOK,TS val 384725297 ecr 0],
length 0
19:58:38.300043 IP (tos 0x10, ttl 64, id 0, offset 0, flags [DF], proto TCP (6),
length 60)
    198.51.100.50.62143 > 172.16.10.10.23: Flags [S], cksum 0x796b (correct), seq
3107170690, win 65535, options [mss 1460,nop,wscale 6,sackOK,TS val 384729498 ecr 0],
length 0
19:58:46.500217 IP (tos 0x10, ttl 64, id 0, offset 0, flags [DF], proto TCP (6),
```



```
length 60)
  198.51.100.50.62143 > 172.16.10.10.23: Flags [S], cksum 0x5964 (correct), seq
3107170690, win 65535, options [mss 1460,nop,wscale 6,sackOK,TS val 384737697 ecr 0],
length 0
^C
```

The source address has been changed from 203.0.113.50 to 198.51.100.50 as per the **ipfw_nat** instance. Note however, that with the configuration binding NAT to an IP address, as opposed to an interface, the NAT aliasing takes place on **all** configured interfaces, internal and external. Verify this by repeating the above tcpdump on bridge1, and running a similar command for an existing host on the internal network. This time the destination sends a TCP reset (RST), since the packet reached the destination but the service on the destination was not open.

```
# telnet 10.10.10.20
```

```
host_system# tcpdump -n -i bridge1 -v
tcpdump: listening on bridge1, link-type EN10MB (Ethernet), snapshot length 262144
bytes
20:12:13.706505 IP (tos 0x10, ttl 64, id 0, offset 0, flags [DF], proto TCP (6),
length 60)
  198.51.100.50.32825 > 10.10.10.20.23: Flags [S], cksum 0x6039 (correct), seq
1314409263, win 65535, options [mss 1460,nop,wscale 6,sackOK,TS val 3924141446 ecr 0],
length 0
20:12:13.710494 IP (tos 0x0, ttl 64, id 0, offset 0, flags [DF], proto TCP (6), length
40)
  10.10.10.20.23 > 198.51.100.50.32825: Flags [R.], cksum 0x5774 (correct), seq 0,
ack 1314409264, win 0, length 0
20:12:29.573756 IP (tos 0x10, ttl 64, id 0, offset 0, flags [DF], proto TCP (6),
length 60)
^C
```

To specify that only the outside interface is to be NATed, use the keyword **if** (interface) on the **ipfw** NAT configuration statement and specify the correct external interface:

```
# ipfw nat 25 config if em1
# ipfw nat show config
ipfw nat 25 config if em1
```

ipfw does not permit the use of **ip ip_addr** and **if interf_name** options at the same time on the same NAT instance. Use one or the other.

What happens in this case is that the NAT instance will ensure that the IP address of interface **em1**

will always be used on traffic exiting through that interface - even if the address changes (because of DHCP or an administrative addressing change):

Traffic destined externally from the **internal** VM host via:

```
root@internal:~ # telnet 172.16.10.10
Trying 172.16.10.10...
^C
```

On the FreeBSD host:

```
host_system# tcpdump -n -i bridge0 -v
tcpdump: listening on bridge0, link-type EN10MB (Ethernet), snapshot length 262144
bytes
20:24:41.147755 IP (tos 0x10, ttl 63, id 0, offset 0, flags [DF], proto TCP (6),
length 60)
    203.0.113.50.40001 > 172.16.10.10.23: Flags [S], cksum 0x5962 (correct), seq
950423268, win 65535, options [mss 1460,nop,wscale 6,sackOK,TS val 2027118491 ecr 0],
length 0
20:24:42.189806 IP (tos 0x10, ttl 63, id 0, offset 0, flags [DF], proto TCP (6),
length 60)
    203.0.113.50.40001 > 172.16.10.10.23: Flags [S], cksum 0x554b (correct), seq
950423268, win 65535, options [mss 1460,nop,wscale 6,sackOK,TS val 2027119538 ecr 0],
length 0
20:24:44.394747 IP (tos 0x10, ttl 63, id 0, offset 0, flags [DF], proto TCP (6),
length 60)
    203.0.113.50.40001 > 172.16.10.10.23: Flags [S], cksum 0x4caa (correct), seq
950423268, win 65535, options [mss 1460,nop,wscale 6,sackOK,TS val 2027121747 ecr 0],
length 0
^C
```

Though it does not look like it, **ipfw** is translating the packets as they exit the firewall.

Consider this exchange where the **internal** VM host pings the **external1** VM:

```
root@internal:~ # ping 203.0.113.10
PING 203.0.113.10 (203.0.113.10): 56 data bytes
64 bytes from 203.0.113.10: icmp_seq=0 ttl=63 time=2.742 ms
64 bytes from 203.0.113.10: icmp_seq=1 ttl=63 time=2.675 ms
^C
```

The traffic on the internal bridge (bridge1) shows the packets from the **internal1** VM:

```

host_system# tcpdump -n -i bridge1 -v
tcpdump: listening on bridge1, link-type EN10MB (Ethernet), snapshot length 262144
bytes
20:29:27.048162 IP (tos 0x0, ttl 64, id 58916, offset 0, flags [none], proto ICMP (1),
length 84)
    10.10.10.20 > 203.0.113.10: ICMP echo request, id 15077, seq 0, length 64
20:29:27.052446 IP (tos 0x0, ttl 63, id 36018, offset 0, flags [none], proto ICMP (1),
length 84)
    203.0.113.10 > 10.10.10.20: ICMP echo reply, id 15077, seq 0, length 64
20:29:28.104133 IP (tos 0x0, ttl 64, id 58917, offset 0, flags [none], proto ICMP (1),
length 84)
    10.10.10.20 > 203.0.113.10: ICMP echo request, id 15077, seq 1, length 64
20:29:28.105732 IP (tos 0x0, ttl 63, id 36019, offset 0, flags [none], proto ICMP (1),
length 84)
    203.0.113.10 > 10.10.10.20: ICMP echo reply, id 15077, seq 1, length 64

```

whereas the traffic on the external bridge (bridge0) shows the correct translation:

```

host_system# tcpdump -n -i bridge0 -v
tcpdump: listening on bridge0, link-type EN10MB (Ethernet), snapshot length 262144
bytes
20:33:19.695939 IP (tos 0x0, ttl 63, id 58919, offset 0, flags [none], proto ICMP (1),
length 84)
    203.0.113.50 > 203.0.113.10: ICMP echo request, id 58206, seq 0, length 64
20:33:19.696546 IP (tos 0x0, ttl 64, id 36021, offset 0, flags [none], proto ICMP (1),
length 84)
    203.0.113.10 > 203.0.113.50: ICMP echo reply, id 58206, seq 0, length 64
20:33:20.715148 IP (tos 0x0, ttl 63, id 58920, offset 0, flags [none], proto ICMP (1),
length 84)
    203.0.113.50 > 203.0.113.10: ICMP echo request, id 58206, seq 1, length 64
20:33:20.715824 IP (tos 0x0, ttl 64, id 36022, offset 0, flags [none], proto ICMP (1),
length 84)
    203.0.113.10 > 203.0.113.50: ICMP echo reply, id 58206, seq 1, length 64
^C

```

The **unreg_only** and **unreg_cgn** configuration options allow bypassing the NAT operation if the source IP of the packet is *not* one of the RFC 1918 addresses (**unreg_only**) or the RFC 6598 addresses (**unreg_cgn** - carrier grade NAT). In these cases, the original source address will be maintained in the packet, even though there is an **ipfw_nat** instance and a matching rule.

```

# ipfw nat 25 show config
ipfw nat 25 config if em1
#
# ipfw nat 25 config if em1 unreg_only
ipfw nat 25 config if em1 unreg_only
#

```

```
# ipfw nat 25 show config
ipfw nat 25 config if em1 unreg_only
#
```

To try the **unreg_only** option, on the **internal** VM, change its IP address on **em0** to a registered number, say 140.140.140.140/24, and change the corresponding link on the firewall (**em1**) to a compatible address - 140.140.140.1/24. The **internal** VM will need a new default route: 140.140.140.1. Ensure that the default route on the **firewall** VM remains 203.0.113.10.

```
root@internal:~ # ifconfig em0 140.140.140.140/24
root@internal:~ # route add default 140.140.140.1
add net default: gateway 140.140.140.1
root@internal:~ #
```

and on the firewall

```
root@firewall:~ # ifconfig em0 140.140.140.1/24
```

From the **internal** VM, try to ping an external address not in the lab:

```
# ping 5.5.5.5
```

and observe on the host system that the **ipfw_nat** instance *did not* replace the source address with the configured IP:

```
host_system# tcpdump -n -i bridge0 -v
tcpdump: listening on bridge0, link-type EN10MB (Ethernet), snapshot length 262144
bytes
21:07:18.154319 IP (tos 0x0, ttl 63, id 58943, offset 0, flags [none], proto ICMP (1),
length 84)
    140.140.140.140 > 5.5.5.5: ICMP echo request, id 38569, seq 0, length 64
21:07:19.180094 IP (tos 0x0, ttl 63, id 58944, offset 0, flags [none], proto ICMP (1),
length 84)
    140.140.140.140 > 5.5.5.5: ICMP echo request, id 38569, seq 1, length 64
21:07:20.194988 IP (tos 0x0, ttl 63, id 58945, offset 0, flags [none], proto ICMP (1),
length 84)
    140.140.140.140 > 5.5.5.5: ICMP echo request, id 38569, seq 2, length 64
```

Not all the options available to **ipfw_nat** are described in the NAT section of the [ipfw\(8\)](#) man page.

Some of the options usable from [natd\(8\)](#) are available to **ipfw_nat**. These include:

```
redirect_port proto targetIP:targetPORT[-targetPORT]
               [aliasIP:]aliasPORT[-aliasPORT]
               [remoteIP[:remotePORT[-remotePORT]]]
redirect_proto proto localIP [publicIP [remoteIP]]

redirect_address localIP publicIP
```

The below options are used for Load Sharing NAT (LSNAT) as described in [RFC 2391](#).

```
redirect_port proto targetIP:targetPORT[,targetIP:targetPORT[,...]]
               [aliasIP:]aliasPORT [remoteIP[:remotePORT]]
redirect_address localIP[,localIP[,...]] publicIP
```

LSNAT is discussed in the next section.

5.3. Setting Up for LSNAT

This example uses the three VMs **external1**, **external2**, and **external3** and pretends they are on the *inside* of the network; and **internal** VM is on the *outside* of the network.

The figure below shows the architecture setup for working with LSNAT.

LSNAT Configuration

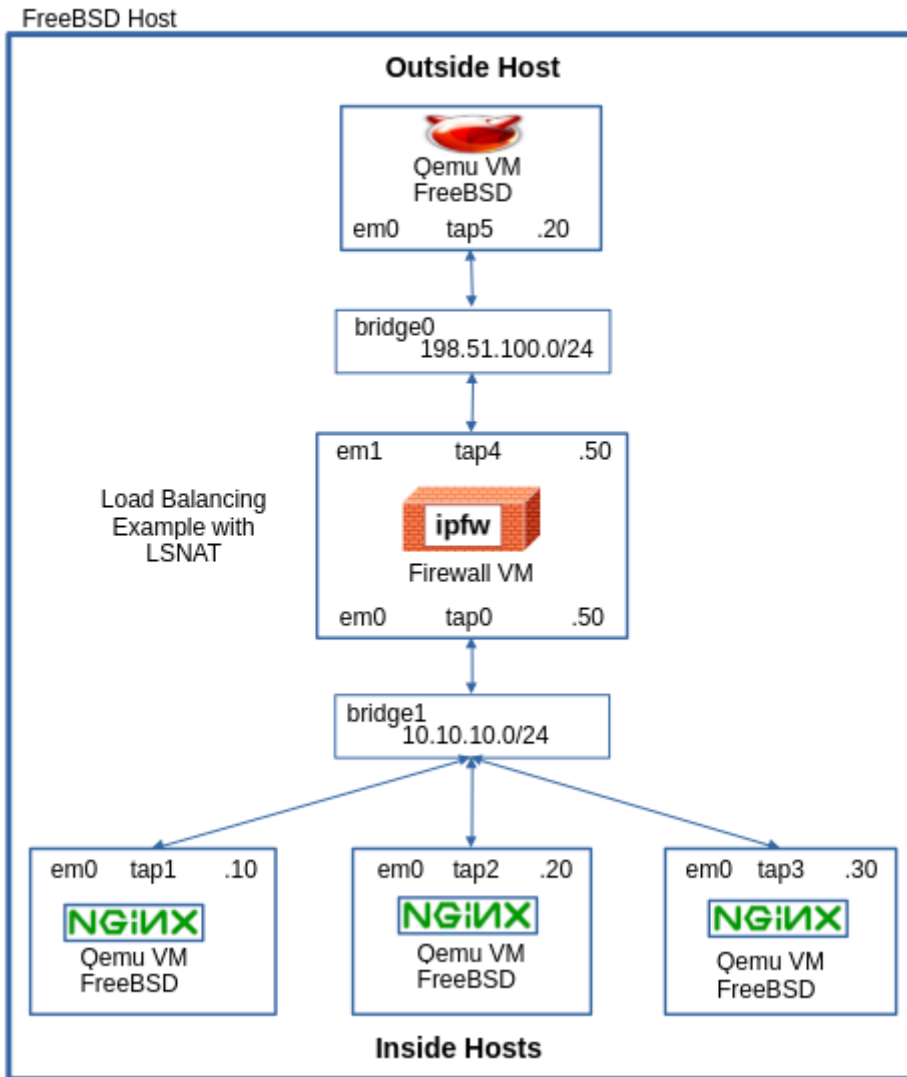


Figure 28. Setting Up for LSNAT

As before, shutdown all virtual machines and rebuild the network from scratch.

Use this command to set up the network bridge and tap architecture.

```
# sudo /bin/sh mkbr.sh reset bridge0 tap4 tap5 bridge1 tap0 tap1 tap2 tap3
```

Note that the host interface is not needed for this example.

Restart the virtual machines with:

```
# /bin/sh runvm.sh firewall internal external1 external2 external3
```

or start them up individually.

Configure each virtual machine to ensure its network configuration matches the above figure and

test connectivity between adjacent systems with `ping(8)`.

Throughout this section, remember that the "external" VMs are now **internal** web servers load balancing between .10, .20, .30, and the "internal" server VM is the **outside** host accessing the internal web servers.

On each **inside** VM the following commands are necessary to perform the examples in this section:

```
# route delete default
#
# route add default 10.10.10.50
```

On the **outside** VM perform these commands:

```
# route delete default
#
# route add default 198.51.100.50
```

Also, on each inside VM, edit the **nginx** `index.html` page and insert a line of text that has the VM name or IP address of the VM - something like this:

```
File: /usr/local/www/nginx/index.html:
```

```
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>
<p> This is VM EXTERNAL1</p>
```

and start **nginx** on each inside VM:

```
# service nginx onestart
Performing sanity check on nginx configuration:
nginx: the configuration file /usr/local/etc/nginx/nginx.conf syntax is ok
nginx: configuration file /usr/local/etc/nginx/nginx.conf test is successful
Starting nginx.
#
```

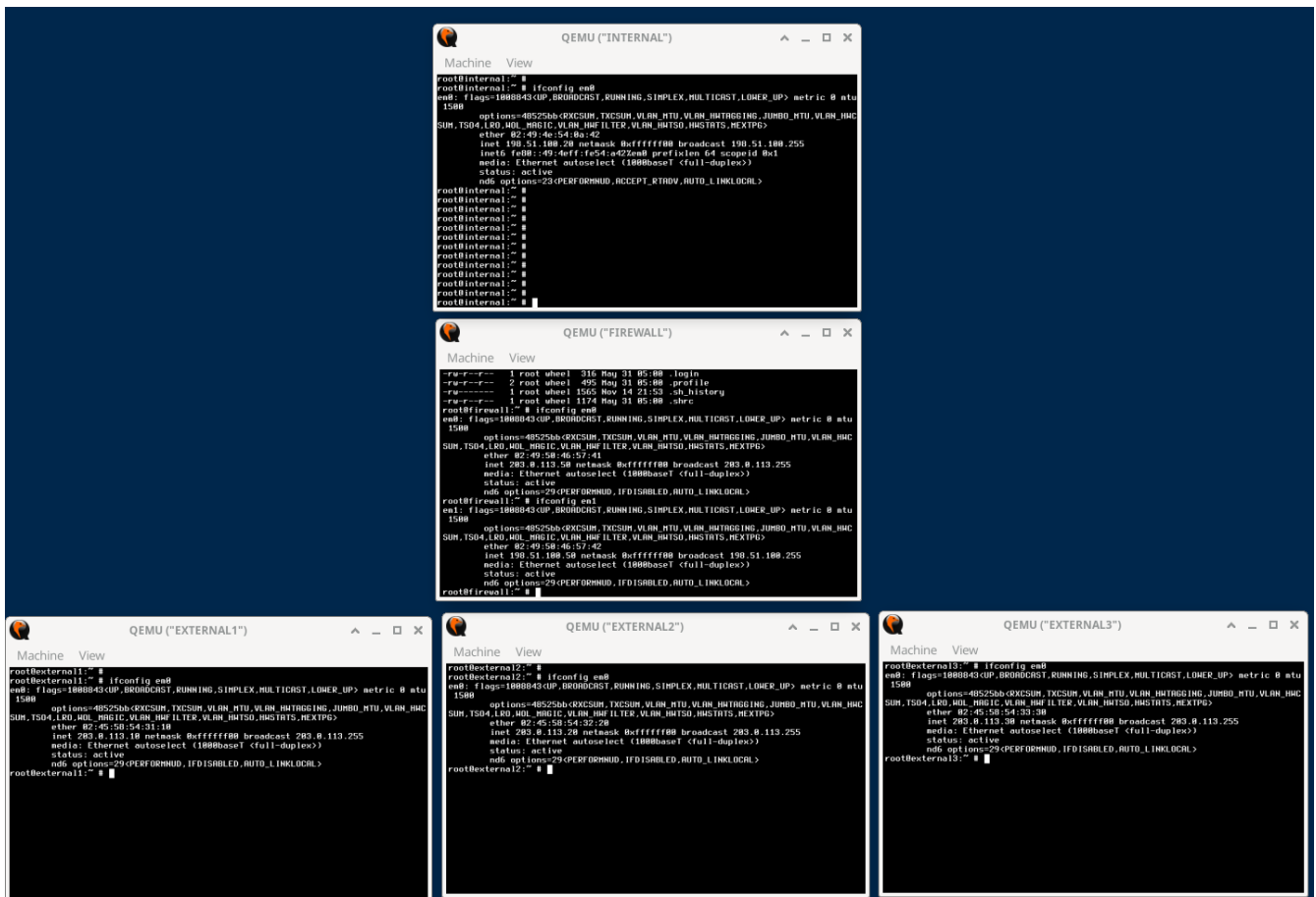


Figure 29. LSNAT Setup Showing All VMs

With no **ipfw** loaded on the firewall, it should be possible to ping all **inside** addresses (10.10.10.10, .20, .30) from the **outside** host (198.51.100.20). And it should be possible to access each web server via:

```
# lynx 10.10.10.10 # (or .20 or .30)
```

5.3.1. Setting up LSNAT- One address (10.10.10.10)

Begin with loading **ipfw** and **ipfw_nat** on the **firewall** VM

```
# kldload ipfw
#
# kldload ipfw_nat
```

The first configuration is similar to static NAT, though from the *outside* to the *inside*. The command redirects incoming traffic from the outside VM sent to destination IP 3.3.3.3 to inside VM 10.10.10.10.


```
# ipfw nat 25 config redirect_addr 10.10.10.10 3.3.3.3
ipfw nat 25 config redirect_addr 10.10.10.10 3.3.3.3
```

Next create a ruleset that utilizes this NAT instance:

```
# ipfw add 50 check-state
# ipfw add 1000 nat 25 tcp from any to any
#
# ipfw list
00050 check-state :default
01000 nat 25 tcp from any to any
65535 deny ip from any to any
#
```



Do not use the **setup** keyword on the **ipfw** rule referencing LSNAT. The **setup** keyword causes the final ACK of the TCP 3-way handshake to be never received and the connection is never established.

From the outside VM, access the web server using:

```
# lynx 3.3.3.3
```

brings up the web page on 10.10.10.10.

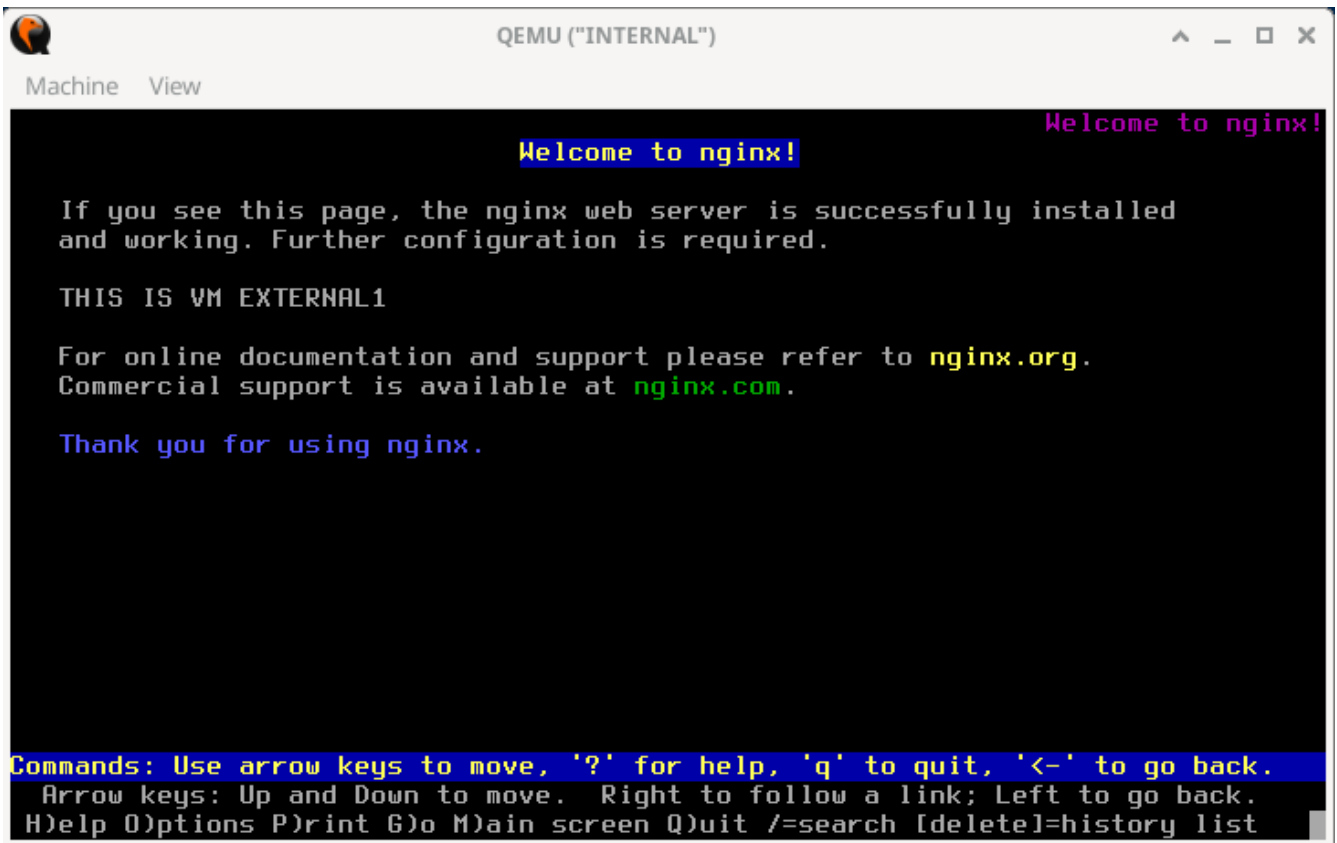


Figure 30. Accessing Nginx on 10.10.10.10 With LSNAT via 3.3.3.3

NAT with one address is working.

5.3.2. Engaging Multiple Hosts With LSNAT

Next, reconfigure the **nat 25** instance to utilize all of the inside hosts:

```
# ipfw nat 25 config redirect_addr 10.10.10.10,10.10.10.20,10.10.10.30 3.3.3.3
```

(Note that adding a modification to a NAT instance just overwrites the existing instance. It does not create a new instance with the same number.)

On the outside VM, running **lynx 3.3.3.3** repeatedly retrieves the home page of each internal server - in round-robin fashion, without regard for any network load, or server utilization.



In the lynx browser, reload the current page by pressing **Ctrl + R**.

```
# ipfw nat 25 show config
ipfw nat 25 config log redirect_addr 10.10.10.10,10.10.10.20,10.10.10.30 3.3.3.3
#
```

By adding a rule to redirect icmp traffic, both icmp and tcp will be load shared across the firewall.

```
# ipfw add 2000 nat 25 icmp from any to any
#
# ipfw list
00050 check-state :default
01000 nat 25 tcp from any to any
02000 nat 25 icmp from any to any
65535 deny ip from any to any
```

Test this by running `tcpdump -n -i em0` on each inside VM, and running `ping -c 1 3.3.3.3` on the outside VM a few times. The incoming ping will hit each inside VM in turn.

However, on running `ping 3.3.3.3`, the result is that these pings hit only one internal VM. The reason is that the aliasing engine treats ICMP differently from TCP and UDP. The aliasing engine recognizes the ICMP id number, and if this number does not change, it uses the same alias. If the command `ping -c 1 3.3.3.3` is used repeatedly, the ICMP id number changes, and this creates a new entry in the aliasing database resulting in redirection to a different VM.

It is common to want to balance the load across servers according to certain characteristics such as system load. This is possible - manually - by reconfiguring the NAT statement and adding multiple instances of the same host to give that host more traffic. Consider this ruleset created with the Unix line continuation character `\` to close the space between successive IP addresses except for the last one and the alias address:

```
# ipfw nat 25 config log redirect_addr \  
10.10.10.30,\  
10.10.10.20,10.10.10.20,\  
10.10.10.10,10.10.10.10,10.10.10.10,10.10.10.10 3.3.3.3
```

This configuration shifts the NAT load heavily toward 10.10.10.10 and moderately toward 10.10.10.20, with 10.10.10.30 having a lot less traffic. Repeat the above single ping example above to see the result. While this works, it is a bit of a hack.

It would be better to have a range assignment feature similar to the sparse address feature already in `ipfw`, something like:

```
# ipfw nat 25 config redirect_addr 10.10.10.0/24{10,20-25,30-50} 3.3.3.3  
ipfw: unknown host 10.10.10.0/24{10
```

but this feature does not work with LSNAT.

However, it is possible to use the `prob` keyword to address load balancing. In a rule with the `prob` keyword, if the rule matches **and** the probability is "true", the action of the rule is taken and processing stops for that packet. If the rule matches, and the probability is "not true", the action is not taken, and processing continues with the next rule. Verify this with a simple test ruleset and the `ucont.sh` shell rule from an external host.

```

03000 prob 0.200000 allow udp from any to me 5656 // set probability to 20% chance of
matching
04000 count udp from any to me // count how many were not chosen by
rule 3000
05000 prob 0.400000 allow udp from any to me 5656 // set probability to 40% chance of
matching
06000 count udp from any to me // count how many were not chosen by
3000 and 5000
07000 prob 0.999000 allow udp from any to me 5656 // set probability to 99.9% chance
of matching
08000 count udp from any to me // count how many were not chosen by
all 3 rules
09000 allow udp from any to me 5656 // unconditional matching
65535 deny ip from any to any // default rule deny

```

After a run of 200 entries from **sh ucont.sh 5656 1** the counts are:

```

03000 47 3314 prob 0.200000 allow udp from any to me 5656
04000 153 10776 count udp from any to me
05000 64 4505 prob 0.400000 allow udp from any to me 5656
06000 89 6271 count udp from any to me
07000 89 6271 prob 0.999000 allow udp from any to me 5656
08000 0 0 count udp from any to me
09000 0 0 allow udp from any to me 5656
65535 0 0 deny ip from any to any

```

From the above data, out of 200 packets sent from **ucont.sh**, 47 were matched by rule 3000, but 153 were not matched (rule 4000). Then, 64 were matched at rule 5000, but 89 were not matched. Finally, 89 were matched at rule 7000.



If there are some packets hitting the default deny rule (65535), delete the host interface from the bridge and re-run the test. **ipfw** is then unlikely to have any stray UDP packets hitting the default rule.

While the above works for UDP, it does not work for TCP. The TCP 3-way handshake is broken because some packets will match, but others will not.

Other load balancing solutions exist for FreeBSD and those should be used instead.

Other NAT Keywords

The other keywords in the NAT section of [ipfw\(8\)](#) are straightforward:

- **deny_in** : deny incoming packets
- **same_ports** : keep the same ports after redirection
- **reset** : clear the aliasing table when the address changes
- **reverse** : reverse the direction of the NAT

- **proxy_only** : packet aliasing is not performed
- **skip_global**
- **global**
- **tablearg** : discussed in [Understanding the Word Tablearg](#)

Chapter 6. IPv6 Network Address Translation (IPv6NAT)

ipfw supports both stateful and stateless IPv6 / IPv4 translation.

From the `ipfw(8)` man page:

Stateful translation

`ipfw` supports in-kernel IPv6/IPv4 network address and protocol translation. Stateful NAT64 translation allows IPv6-only clients to contact IPv4 servers using unicast TCP, UDP or ICMP protocols. One or more IPv4 addresses assigned to a stateful NAT64 translator are shared among several IPv6-only clients. When stateful NAT64 is used in conjunction with DNS64, no changes are usually required in the IPv6 client or the IPv4 server. The kernel module `ipfw_nat64` should be loaded or kernel should have options `IPFIREWALL_NAT64` to be able use stateful NAT64 translator.

Stateful translation is suitable for deployment at the client side or at the service provider, allowing IPv6-only client hosts to reach remote IPv4-only nodes.

Stateless translation is appropriate when a NAT64 translator is used in front of IPv4-only servers to allow them to be reached by remote IPv6-only clients.

Specific requirements for these translation services are found in a collection of RFCs:

- Stateful NAT64: Network Address and Protocol Translation from IPv6 Clients to IPv4 Servers - [RFC 6146](#)
- IPv6 Addressing of IPv4/IPv6 Translators - [RFC 6052](#)
- IPv6 Address Prefix Reserved for Documentation - [RFC 3849](#)

There are a couple of bugs registered for NAT64. See the following: NAT64 https://bugs.freebsd.org/bugzilla/show_bug.cgi?id=255928 (NAT64 issue on 13.0)

The lab examples for all IPv6 / IPv4 translations will use two new virtual machines:

- **dnshost** - this virtual machine runs a configured copy of BIND 9. Some experience with DNS setup with BIND 9 is helpful but not required.
- **v6only** - this virtual machine only uses IPv6. It is not configured for IPv4 addressing at all.

Readers should have a basic understanding of IPv6 and its addressing characteristics. These resources may be helpful:

- https://en.wikipedia.org/wiki/IPv6_transition_mechanism
- https://en.wikipedia.org/wiki/IPv6_address

6.1. Stateful NAT64 (NAT64LSN) With DNS64

NAT64, described in RFC 6146 is one of a number of transition mechanisms that companies can take as they introduce IPv6 into their environment, or move wholesale into IPv6 locally. The idea with NAT64 is to provide a mechanism to allow an IPv6-only host to make a connection to a remote IPv4-only host. This includes the ability to do a DNS lookup on the remote host, and through the features of DNS64 (a companion transition service described in RFC 6147), translate a received IPv4 address into a special IPv6 address that provides a way to connect using the Network Address Translation variant called NAT64.

A logical view of NAT64 and DNS64 is shown in the figure below:

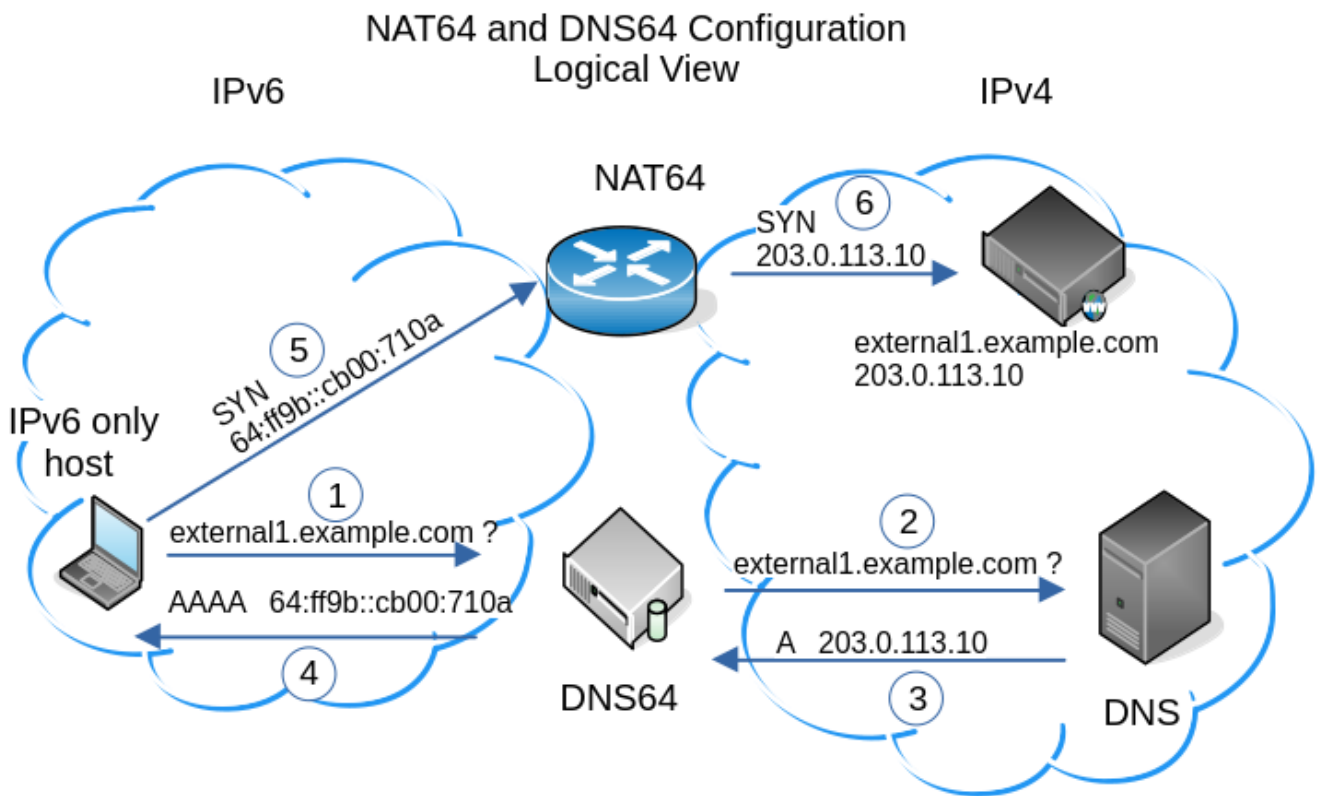


Figure 31. Logical View of NAT64 and DNS64

The process works like this:

1. An IPv6 only host wants to access a resource from host **external1.example.com** which only uses IPv4. A DNS lookup for "external1.example.com" is sent to the locally configured DNS64 server. This lookup is for an "AAAA" record for the external1 VM.
2. The DNS64 server forwards the request to an authoritative server for "example.com".
3. The authoritative server returns an IPv4 address back to the DNS64 server.
4. The DNS64 server converts the IPv4 address into an IPv6 address using the transition service described in RFC 6147.
5. The IPv6 only host, receives the IPv6 address and sends a connection request (SYN) to its local IPV6 router running NAT64.
6. The NAT64 router converts the IPv6 packet back to IPv4 and forwards the packet to external1.example.com.

The remaining conversions between the IPv6 VM and external1 VM happen in a similar fashion.

In step 4, the DNS64 server converts the IPv4 address into IPv6 by using the "Well Known Prefix" `64:ff9b::` and encapsulating the IPv6 address into the last 4 octets of the address. In the figure above, "203.0.113.10" has been converted to "cb00:710a" and added as the last four octets of the new address.

Note that this is one instance of a larger selection of translation algorithms to translate an IPv4 address into an IPv6 address. In this implementation, the DNS64 server and the authoritative server are essentially merged together following the description of "Example of 'an IPv6 Network to the IPv4 Internet' Setup with DNS64 in Stub-Resolver Mode" in Section 7.2 of RFC 6147.

6.1.1. Setting Up for NAT64 / DNS64

To exercise the NAT64 capabilities of `ipfw`, it is first necessary to restart all lab virtual machines and reconfigure the `ipfw` lab.

The figure below shows the new configuration needed.

NAT64 and DNS64 Configuration

FreeBSD Host

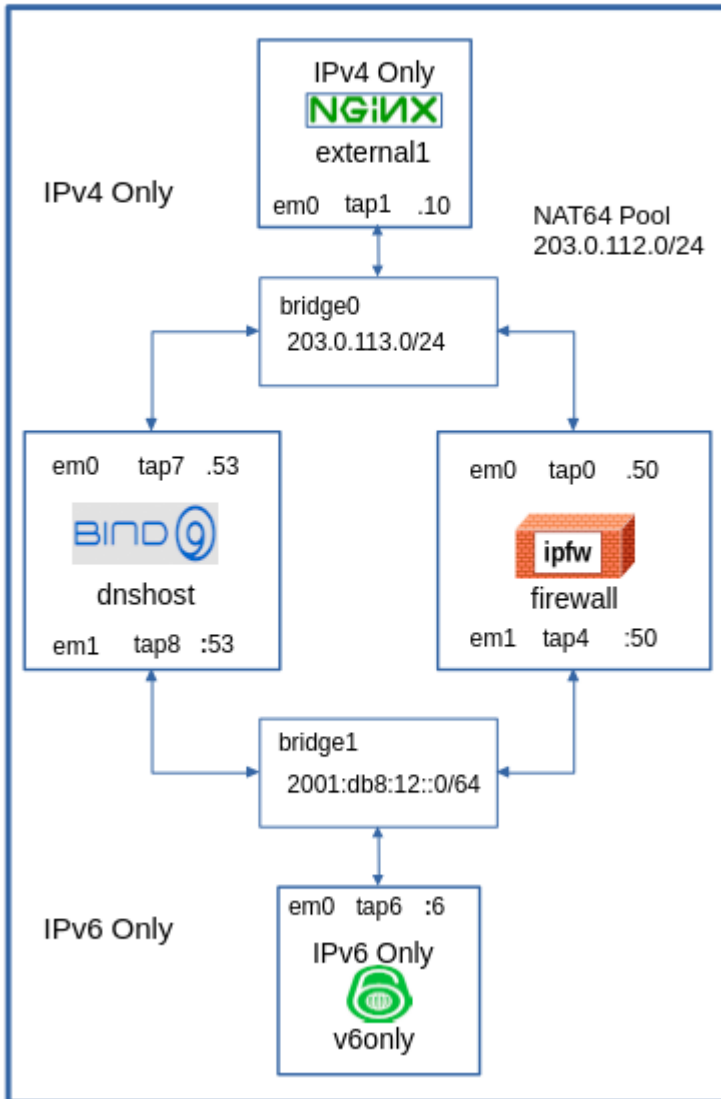


Figure 32. Network Setup for NAT64 and DNS64 Examples

On the FreeBSD host system, the appropriate bridge and tap setup is given by this command:

```
$ sudo /bin/sh mkbr.sh reset bridge0 tap0 tap1 tap7 bridge1 tap4 tap6 tap8
```

Start up the required virtual machines with:

```
$ /bin/sh runvm.sh firewall dnshost external1 v6only
```

As before, configure all interfaces and ensure connectivity of adjacent interfaces. The **firewall** VM should be set for both IPv4 forwarding and IPv6 forwarding:

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

The **external1** VM default route should point to 203.0.113.50 and the v6only host default IPv6 route should point to 2001:db8:12::50 as follows:

On external1:

```
# route add default 203.0.113.50
```

On v6only:

```
# route -6 add default 2001:db8:12::50
```



RFC 5737 describes the use of the 203.0.113.0/24 network for documentation and example purposes. RFC 3849 describes the use of the 2001:db8::/32 network for the same purposes.

6.1.2. Setting Up the dnshost VM

First, set up the **dnshost** VM to provide DNS64 services. ISC's bind9 (9.18 and above) provides this capability. Setting up bind9, while not trivial, is not impossible. Install the following packages:

- **bind9** Use the latest supported version. The server running here is using bind 9.20.5
- **bind-tools** Same note as above. The tools used here are bind-tools-9.20.5

These packages will install a modest number of dependencies.

Included with the VM_SCRIPTS, copy the `.tgz` file `~/ipfw-primer/ipfw/VM_SCRIPTS/dnshost/dnshost_usrlocaletc_namedb.tgz` for the bind9 configuration files needed. Use the following commands to retrieve and untar the files:

```
# scp user@host:~/ipfw-primer/ipfw/VM_SCRIPTS/dnshost/dnshost_usrlocaletc_namedb.tgz .  
# tar xvzf dnshost_usrlocaletc_namedb.tgz -C /usr/local/etc
```

This will install all the needed DNS files. Otherwise see the zone files in [Appendix E](#), and try to set up DNS. Note that the files include a stub root zone. This provides a locally complete DNS setup.

Start the **named** service with:

```
# service named onestart
```

There should not be any errors, but if there are, track down and fix.

Test the dnshost configuration with these commands. The first lookup returns the A resource record with an IPv4 address. The second lookup returns the AAAA resource record with the DNS64

configured "Well-known Prefix" 64:ff9b that is used in this section.

```
root@dnshost:~ # dig @localhost external1.example.com

; <<>> DiG 9.16.27 <<>> @localhost external1.example.com
; (2 servers found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 61764
;; flags: qr aa rd; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1
;; WARNING: recursion requested but not available

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 1232
; COOKIE: 96f573e1a62ce4380100000062884cacf9b86d6e9f54b542 (good)
;; QUESTION SECTION:
;external1.example.com.          IN      A

;; ANSWER SECTION:
external1.example.com.  3600    IN      A      203.0.113.10

;; Query time: 58 msec
;; SERVER: ::1#53(::1)
;; WHEN: Fri May 20 22:21:32 EDT 2022
;; MSG SIZE rcvd: 94
#
#
root@dnshost:~ # dig @localhost external1.example.com aaaa

; <<>> DiG 9.16.27 <<>> @localhost external1.example.com aaaa
; (2 servers found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 5865
;; flags: qr aa rd; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1
;; WARNING: recursion requested but not available

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 1232
; COOKIE: 0ad6c60faab12f7f0100000062884cb0be6e6bd00f7e482f (good)
;; QUESTION SECTION:
;external1.example.com.          IN      AAAA

;; ANSWER SECTION:
external1.example.com.  3600    IN      AAAA   64:ff9b::cb00:710a
#
```

The last test is the most important. The dnshost must return the Well-Known Prefix "64:ff9b::" with

the corresponding bits for the embedded IPv4 address. If the test does not return this value, reconfigure the DNS service (named.conf and the primary forward zone "example.com") to fix.

Since the **v6only** machine will only ever request AAAA lookups, the configuration is complete for this section. Ensure that the /etc/resolv.conf file on the **v6only** VM is correctly configured:

```
root@v6only: # cat /etc/resolv.conf
nameserver 2001:db8:12::53

root@v6only: #
```

Next to set up are the **firewall**, the IPV6 only host **v6only**, and the external VM host, **external1**. Test connectivity without any **ipfw** running on the **firewall** host. Refer to the above diagram for network addressing.

Finally, proceed with the installation of NAT64 on the **ipfw firewall**.

On the firewall VM:

```
# kldload ipfw

The next line loads the NAT64 module.

# kldload ipfw_nat64
```

Configuring NAT64 is similar to configuring NAT. Create an instance of the NAT64 translator first.

```
# ipfw nat64lsn foo create prefix4 203.0.112.0/24 allow_private
```

The use of the "**allow_private**" keyword is required. The [ipfw\(8\)](#) manual page notes that the NAT64 translator, by default, will not handle addresses whose destination matches those listed in RFC 1918. The addressing scheme in this lab uses special purpose addresses as noted in RFC 6890 which are **also** considered "private addresses" by the ipfw NAT64 translator.

Note that the prefix4 address pool (**203.0.112.0/24** above) should not be manually configured as an alias on any interface. These addresses are used internally by **ipfw**. The only requirement is that they be reserved from deployment elsewhere in the local network so they do not cause a routing conflict with **ipfw**. This allows for 254 simultaneous NAT64 addresses. If more are needed due to high volume, add another prefix4, or increase the existing prefix4 address space.

Continue configuring the NAT64 / DNS64 translator:

```
# ipfw add allow log ipv6-icmp from any to any icmp6types 135,136
```

```
# ipfw add nat64lsn foo log ip from 2001:db8:12::/64 to 64:ff9b::/96 in
# ipfw add nat64lsn foo log ip from any to 203.0.112.0/24 in
# ipfw add allow log ip from any to any
```

and the `direct_output` sysctl must be set to 1 (not zero):

```
# sysctl net.inet.ip.fw.nat64_direct_output=1
```

If desired, also set the `nat64_debug` sysctl and the `firewall verbose` sysctl:

```
# sysctl net.inet.ip.fw.nat64_debug=1
# sysctl net.inet.ip.fw.verbose=1
```

Use `tail -f /var/log/security` to view the `nat64lsn` translations.

With these prerequisites completed the following tests on the **v6only** VM should be successful:

```
root@v6only# ping6 -c 3 64:ff9b::203.0.113.10
PING6(56=40+8+8 bytes) 2001:db8:12::30 --> 64:ff9b::cb00:710a
16 bytes from 64:ff9b::cb00:710a, icmp_seq=0 hlim=63 time=8.401 ms
16 bytes from 64:ff9b::cb00:710a, icmp_seq=1 hlim=63 time=3.429 ms
16 bytes from 64:ff9b::cb00:710a, icmp_seq=2 hlim=63 time=3.398 ms

--- 64:ff9b::203.0.113.10 ping6 statistics ---
3 packets transmitted, 3 packets received, 0.0% packet loss
round-trip min/avg/max/std-dev = 3.398/5.076/8.401/2.351 ms
```

and using `lynx` to grab the **external1.example.com** home page should be successful:

```
[root@v6only ~]# lynx external1.example.com
```

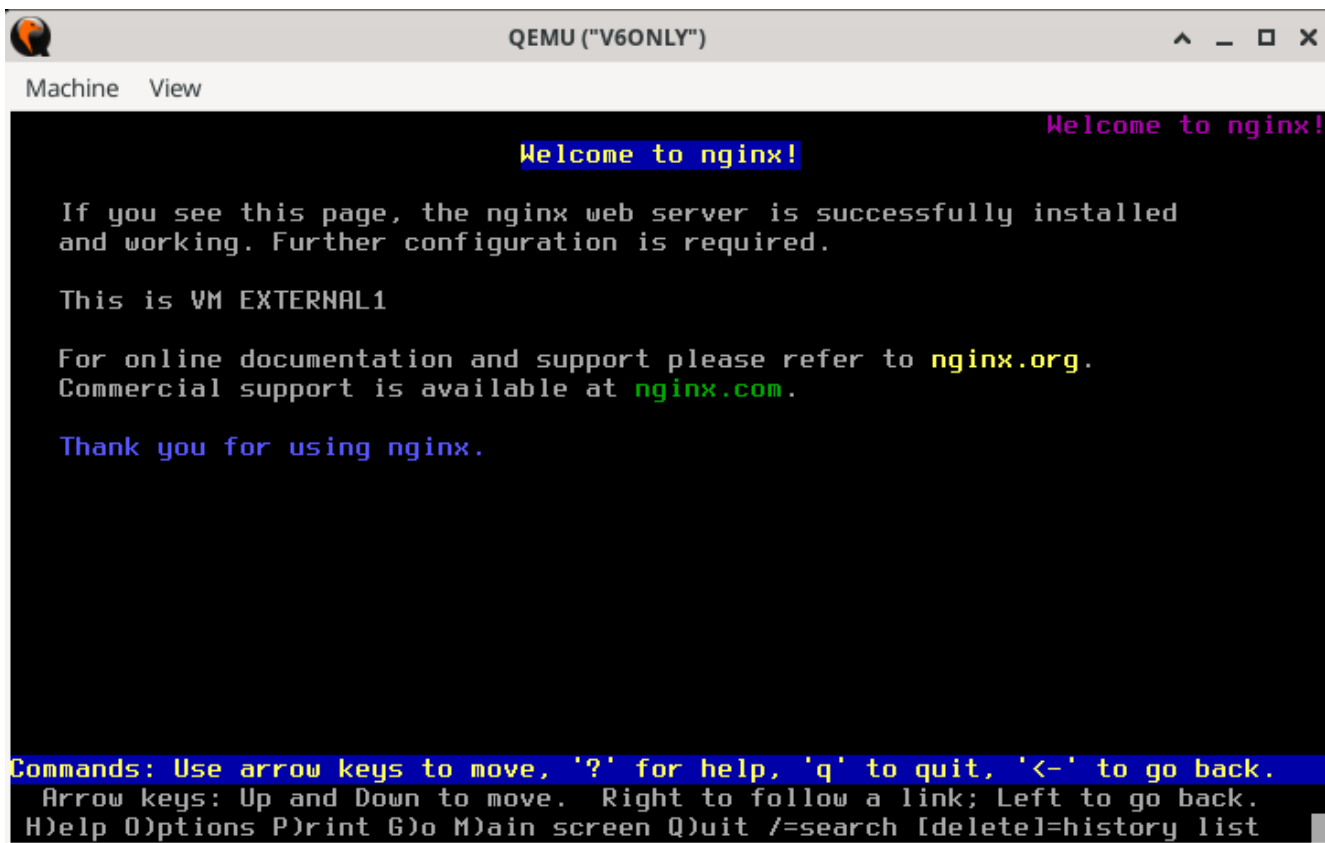


Figure 33. Viewing the IPv4 Webpage of external1 VM from IPv6 v6only VM.

6.1.3. Setting Up for Stateless NAT64 - NAT64STL

The previous `ipfw_nat64` examples used "stateful" address translation. `ipfw` is also capable of performing "stateless" address translation.

Stateless translation is appropriate when a NAT64 translator is used in front of IPv4-only servers to allow them to be reached by remote IPv6-only clients. Stateful translation is suitable for deployment at the client side or at the service provider, allowing IPv6-only client hosts to reach remote IPv4-only nodes.

Stateless configuration of NAT64 is possible with the same architecture as the previous stateful example. Configuration details however, are different. In the stateless case, `ipfw` uses two tables for translating addresses in either direction: **IPv4** → **IPv6** and **IPv6** → **IPv4**. A typical configuration is shown below.

```
Start fresh
# kldunload ipfw_nat64
# kldunload ipfw
# kldload ipfw
# kldload ipfw_nat64
```

Create the tables used for ipfw_nat64stl

```
# ipfw table T4to6 create type addr valtype ipv6
# ipfw table T6to4 create type addr valtype ipv4
# ipfw table T4to6 add 203.0.112.1 2001:db8:12::6
# ipfw table T6to4 add 2001:db8:12::6 203.0.112.1
# ipfw nat64stl NAT64 create table4 T4to6 table6 T6to4 allow_private
```

Add rules for ipfw_nat64stl

```
# ipfw add allow log icmp6 from any to any icmp6types 135, 136
# ipfw add nat64stl NAT64 log ip from any to 'table(T4to6)'
# ipfw add nat64stl NAT64 log ip6 from 'table(T6to4)' to 64:ff9b::/96
# ipfw add allow log ip from any to any
```

Adjust sysctls

```
# sysctl net.inet.ip.fw.verbose=1
# sysctl net.inet.ip.fw.nat64_debug=1
# sysctl net.inet.ip.fw.nat64_direct_output=1
```

Only the **net.inet.ip.fw.nat64_direct_output** sysctl is required.

Use the same tests as in the stateful NAT64 example:

```
[root@v6only ~]# ping6 -c 3 64:ff9b::203.0.113.10
and
[root@v6only ~]# lynx external1.example.com
```

Both tests should be successful.

It may seem limiting to have to use tables to effect communication for stateless NAT64. However, in considering the architecture involved, the above statements about stateless translation being

appropriate when a NAT64 translator is used in front of IPv4-only servers to allow them to be reached by remote IPv6-only clients makes sense. The entire IPv6 cloud can reach a specified server.

This can be accomplished by, for example, changing the T4to6 and T6to4 tables to read:

```
# ipfw table T4to6 add 203.0.112.0/31      2001:db8:12::30
# ipfw table T6to4 add 2000:0000:0000::/8  203.0.112.0
# ipfw table T6to4 add 2100:0000:0000::/8  203.0.112.1
```

The T4to6 table allocates two addresses in the address pool: 203.0.112.0 and 203.0.112.1. These are used separately in the T6to4 table to cover vast ranges of IPv6 address space.

Certainly using just one IPv4 pool address is not going to be sufficient to translate such a large range of IPv6 addresses. The point here is that by carefully constructing the translation pool addresses and the T4to6 and T6to4 address tables, **ipfw** can manage translation to as many IPv6 addresses as needed.

Note that stateless NAT64 shares the same limitations of stateful NAT64.

Next is the most important IPv6 / IPv4 translation mechanism NAT64 CLAT.

6.2. 464XLAT

ipfw supports 464XLAT (RFC 6877) calling it "XLAT464 CLAT". This transition mechanism provides connectivity for IPv4 edge devices across an IPv6 only network. It does this by combining stateful translation in the core and stateless translation at the edge. 464XLAT only supports IPv4 in the client-server model, so it does not support IPv4 peer-to-peer communication or inbound IPv4 connections.



See diagrams and explanation here: <https://www.juniper.net/documentation/us/en/software/junos/interfaces-adaptive-services/topics/topic-map/ipv4-connect-ipv6-464xlat.html#id-464xlat-overview>

The RFC for this mechanism is more enlightening. See [RFC 6877](#)

The discussion on Wikipedia is somewhat sparse:

464XLAT

464XLAT (RFC 6877) allows clients on IPv6-only networks to access IPv4-only Internet services, such as Skype.[13][14]

The client uses a SIIT translator to convert packets from IPv4 to IPv6. These are then sent to a NAT64 translator which translates them from IPv6 back into IPv4 and on to an IPv4-only server. The client translator may be implemented on the client itself or on an intermediate device and is known as the CLAT (Customer-side transLATOR). The NAT64 translator, or PLAT (Provider-side transLATOR), must be able to reach both the server and the client (through the CLAT). The use of NAT64 limits connections to a client-server model using UDP, TCP, and ICMP.

The figure below shows a diagram for implementing 464XLAT.

464XLAT Configuration

FreeBSD Host

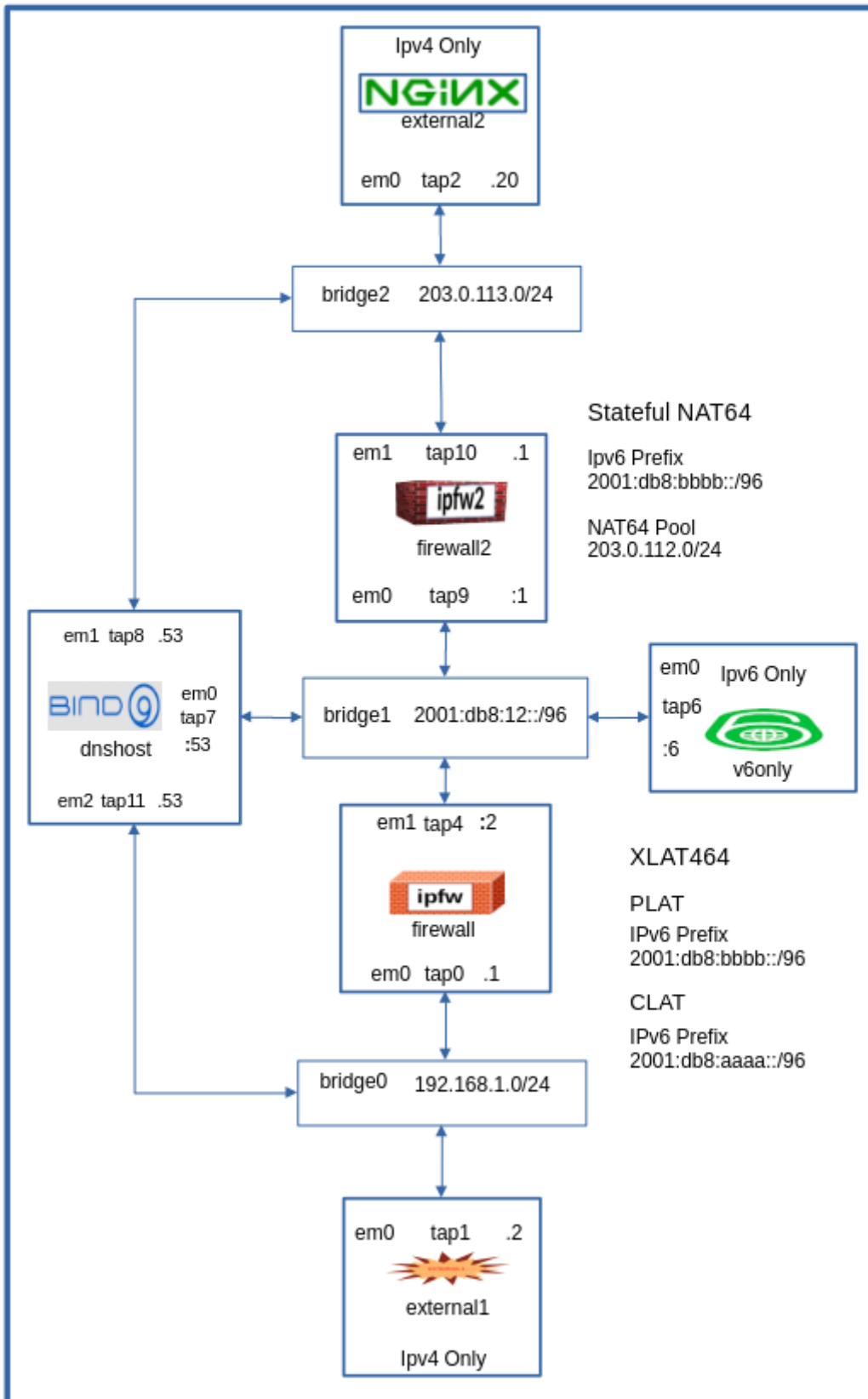


Figure 34. ipfw 464XLAT Design

As earlier, shutdown all virtual machines and for this example, reset all the bridge and tap devices to the new architecture.

This example will require two firewalls. The **firewall** VM and **firewall2** VM will both be used as

shown in the diagram. In this example, the **firewall** VM is the CLAT translator (stateless translation) and the **firewall2** VM is the PLAT translator (stateful translation).

To start, set up the bridge and tap interfaces with this command on the FreeBSD host:

```
$ sudo /bin/sh mkbr.sh reset bridge0 tap0 tap1 tap11 bridge1 tap4 tap6 tap7 tap9 \  
bridge2 tap2 tap8 tap10
```

Start up the virtual machines with:

```
$ /bin/sh runvm.sh external1 firewall firewall2 dnshost v6only external2
```

As earlier, configure all interfaces according to the diagram and ensure connectivity with adjacent interfaces.

This example is more complex than past examples. There are a number of additional configuration steps needed as follows:

- **external1.example.com**

```
On external1.example.com:
```

```
route add default 192.168.1.1  
echo "nameserver 192.168.1.53" > /etc/resolv.conf  
echo "nameserver 203.0.113.53" >> /etc/resolv.conf
```

- **firewall.example.com**

```
On firewall.example.com:
```

```
/bin/sh /root/bin/bsdclat464.sh  
echo "nameserver 2001:db8:12::53" > /etc/resolv.conf  
echo "nameserver 192.168.1.53" >> /etc/resolv.conf  
route -6 add 2001:db8:bbbb::/96 2001:db8:12::1  
sysctl net.inet.ip.forwarding=1  
sysctl net.inet6.ip6.forwarding=1  
sysctl net.inet.ip.fw.verbose=1  
sysctl net.inet.ip.fw.nat64_direct_output=1
```

- **firewall2.example.com**

```
On firewall2.example.com
```

```
/bin/sh /root/bin/bsdplat464.sh
```

```
route -6 add 2001:db8:aaaa::/96 2001:db8:12::2
echo "nameserver 2001:db8:12::53" > /etc/resolv.conf
sysctl net.inet.ip.forwarding=1
sysctl net.inet6.ip6.forwarding=1
sysctl net.inet.ip.fw.verbose=1
sysctl net.inet.ip.fw.nat64_direct_output=1
```

- **external2.example.com**

On external2.example.com

```
route add default 203.0.113.1
echo "nameserver 203.0.113.53" > /etc/resolv.conf
service nginx onestart
```

- **dnshost.example.com**

The DNS64 capability is not used for this example.

On dnshost.example.com:

Remove DNS64:

```
vi /usr/local/etc/namedb/named.conf
```

Comment out the dns64 clause in the options section:

```
. . .
//      dns64 64:FF9B::/96 {
//      clients { any; };
//      exclude { 64:FF9B::/96; ::ffff:0000:0000/96; };
//      suffix ::;
//      };
. . .
```

```
echo "nameserver 127.0.0.1" > /etc/resolv.conf
echo "nameserver 2001:db8:12::53" >> /etc/resolv.conf
service named onestart
route add default 203.0.113.1
route add -net 192.168.1.0/24 192.168.1.1
sysctl net.inet.ip.forwarding=0
sysctl net.inet6.ip6.forwarding=0
```

- **v6only.example.com**

On v6only.example.com:

```
echo "nameserver 2001:db8:12::53" > /etc/resolv.conf
route -6 add 2001:db8:bbbb::/96 2001:db8:12::1
route -6 add 2001:db8:aaaa::/96 2001:db8:12::2
```

Due to the complex nature of these **ipfw** configurations, the **bsdclat464.sh** and **bsdplat464.sh** scripts are provided. However, do try to understand the configuration details.

Once all the above commands are entered on their respective VMs, test the configuration with a ping from the **external1** VM to the **external2** VM:

```
root@external1:# ping -c 2 external2.example.com
PING external2.example.com (203.0.113.20): 56 data bytes
64 bytes from 203.0.113.20: icmp_seq=0 ttl=62 time=5.578 ms
64 bytes from 203.0.113.20: icmp_seq=1 ttl=62 time=8.002 ms

--- external2.example.com ping statistics ---
2 packets transmitted, 2 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 5.578/6.790/8.002/1.212 ms
root@external1:#
```

While the above results look unremarkable, what has actually happened is that an IPV4 host (the **external1** VM) initiated an IPV4 communication (**ping**) to the IPV4 **external2** VM. This communication was translated into IPV6 by the edge **firewall** VM and forwarded over the IPV6 network to the **firewall2** VM which translated it back to IPv4 and forwarded it to the **external1** VM. The IPV4 reply took a similar path, getting translated into IPV6 and routed over the IPV6 network. The packet was finally translated back to IPv4 by the edge device, **firewall** VM.

The snippets below show at each step, how the request was transformed.



The examples below are taken from multiple different invocations of the **ping** command. However, the data transformations are correct.

On interface **em0** on the **firewall** VM:

```
root@firewall:~/bin # tcpdump -n -i em0
tcpdump: verbose output suppressed, use -v[v]... for full protocol decode
listening on em0, link-type EN10MB (Ethernet), snapshot length 262144 bytes
14:38:07.254114 IP 192.168.1.2 > 203.0.113.20: ICMP echo request, id 46395, seq 0,
length 64
14:38:07.256893 IP 203.0.113.20 > 192.168.1.2: ICMP echo reply, id 46395, seq 0,
length 64
14:38:08.322597 IP 192.168.1.2 > 203.0.113.20: ICMP echo request, id 46395, seq 1,
length 64
```

```
14:38:08.326715 IP 203.0.113.20 > 192.168.1.2: ICMP echo reply, id 46395, seq 1, length 64
```

On interface **em1** on the **firewall** VM:

```
root@firewall:~/bin # tcpdump -n -i em1
tcpdump: verbose output suppressed, use -v[v]... for full protocol decode
listening on em1, link-type EN10MB (Ethernet), snapshot length 262144 bytes
14:54:45.140746 IP6 2001:db8:aaaa::c0a8:102 > 2001:db8:bbbb::cb00:7114: ICMP6, echo request, id 38233, seq 0, length 64
14:54:45.142995 IP6 2001:db8:bbbb::cb00:7114 > 2001:db8:aaaa::c0a8:102: ICMP6, echo reply, id 38233, seq 0, length 64
14:54:46.171754 IP6 2001:db8:aaaa::c0a8:102 > 2001:db8:bbbb::cb00:7114: ICMP6, echo request, id 38233, seq 1, length 64
14:54:46.173925 IP6 2001:db8:bbbb::cb00:7114 > 2001:db8:aaaa::c0a8:102: ICMP6, echo reply, id 38233, seq 1, length 64
```

On interface **em0** on the **firewall2** VM:

```
root@firewall2:~/bin # tcpdump -n -i em0
tcpdump: verbose output suppressed, use -v[v]... for full protocol decode
listening on em0, link-type EN10MB (Ethernet), snapshot length 262144 bytes
14:57:32.519334 IP6 2001:db8:aaaa::c0a8:102 > 2001:db8:bbbb::cb00:7114: ICMP6, echo request, id 17270, seq 0, length 64
14:57:32.529066 IP6 2001:db8:bbbb::cb00:7114 > 2001:db8:aaaa::c0a8:102: ICMP6, echo reply, id 17270, seq 0, length 64
14:57:33.560392 IP6 2001:db8:aaaa::c0a8:102 > 2001:db8:bbbb::cb00:7114: ICMP6, echo request, id 17270, seq 1, length 64
14:57:33.561596 IP6 2001:db8:bbbb::cb00:7114 > 2001:db8:aaaa::c0a8:102: ICMP6, echo reply, id 17270, seq 1, length 64
```

On interface **em1** on the **firewall2** VM:

```
root@firewall2:~/bin # tcpdump -n -i em1
tcpdump: verbose output suppressed, use -v[v]... for full protocol decode
listening on em1, link-type EN10MB (Ethernet), snapshot length 262144 bytes
14:58:37.139612 IP 203.0.112.22 > 203.0.113.20: ICMP echo request, id 1025, seq 0, length 64
14:58:37.141043 IP 203.0.113.20 > 203.0.112.22: ICMP echo reply, id 1025, seq 0, length 64
14:58:38.187477 IP 203.0.112.22 > 203.0.113.20: ICMP echo request, id 1025, seq 1, length 64
14:58:38.188308 IP 203.0.113.20 > 203.0.112.22: ICMP echo reply, id 1025, seq 1, length 64
```

On interface **em0** on the **external2** VM:

```
root@external2:~ # tcpdump -n -i em0
tcpdump: verbose output suppressed, use -v[v]... for full protocol decode
listening on em0, link-type EN10MB (Ethernet), snapshot length 262144 bytes
15:00:44.171439 IP 203.0.112.22 > 203.0.113.20: ICMP echo request, id 1024, seq 0,
```

```
length 64
15:00:44.172313 IP 203.0.113.20 > 203.0.112.22: ICMP echo reply, id 1024, seq 0,
length 64
15:00:45.200883 IP 203.0.112.22 > 203.0.113.20: ICMP echo request, id 1024, seq 1,
length 64
15:00:45.201035 IP 203.0.113.20 > 203.0.112.22: ICMP echo reply, id 1024, seq 1,
length 64
```

The firewall log `sysctl` was reset to log to `syslogd(8)` and captured these logs:

Logs on the `firewall` VM:

```
root@firewall:~/bin # cat /var/log/security
Dec  2 15:14:04 firewall kernel: ipfw: 150 Eaction nat64clat ICMP:8.0 192.168.1.2
203.0.113.20 in via em0
Dec  2 15:14:04 firewall kernel: ipfw: 150 Eaction nat64clat ICMPv6:129.0
[2001:db8:bbbb::cb00:7114] [2001:db8:aaaa::c0a8:102] in via em1
Dec  2 15:14:04 firewall kernel: ipfw: 150 Eaction nat64clat ICMP:8.0 192.168.1.2
203.0.113.20 in via em0
Dec  2 15:14:05 firewall kernel: ipfw: 150 Eaction nat64clat ICMPv6:129.0
[2001:db8:bbbb::cb00:7114] [2001:db8:aaaa::c0a8:102] in via em1
Dec  2 15:14:08 firewall kernel: ipfw: 100 Accept ICMPv6:135.0 [2001:db8:12::1]
[2001:db8:12::2] in via em1
Dec  2 15:14:08 firewall kernel: ipfw: 100 Accept ICMPv6:136.0 [2001:db8:12::2]
[2001:db8:12::1] out via em1
Dec  2 15:14:09 firewall kernel: ipfw: 100 Accept ICMPv6:135.0 [2001:db8:12::2]
[2001:db8:12::1] out via em1
Dec  2 15:14:09 firewall kernel: ipfw: 100 Accept ICMPv6:136.0 [2001:db8:12::1]
[2001:db8:12::2] in via em1
```

Logs on the `firewall2` VM:

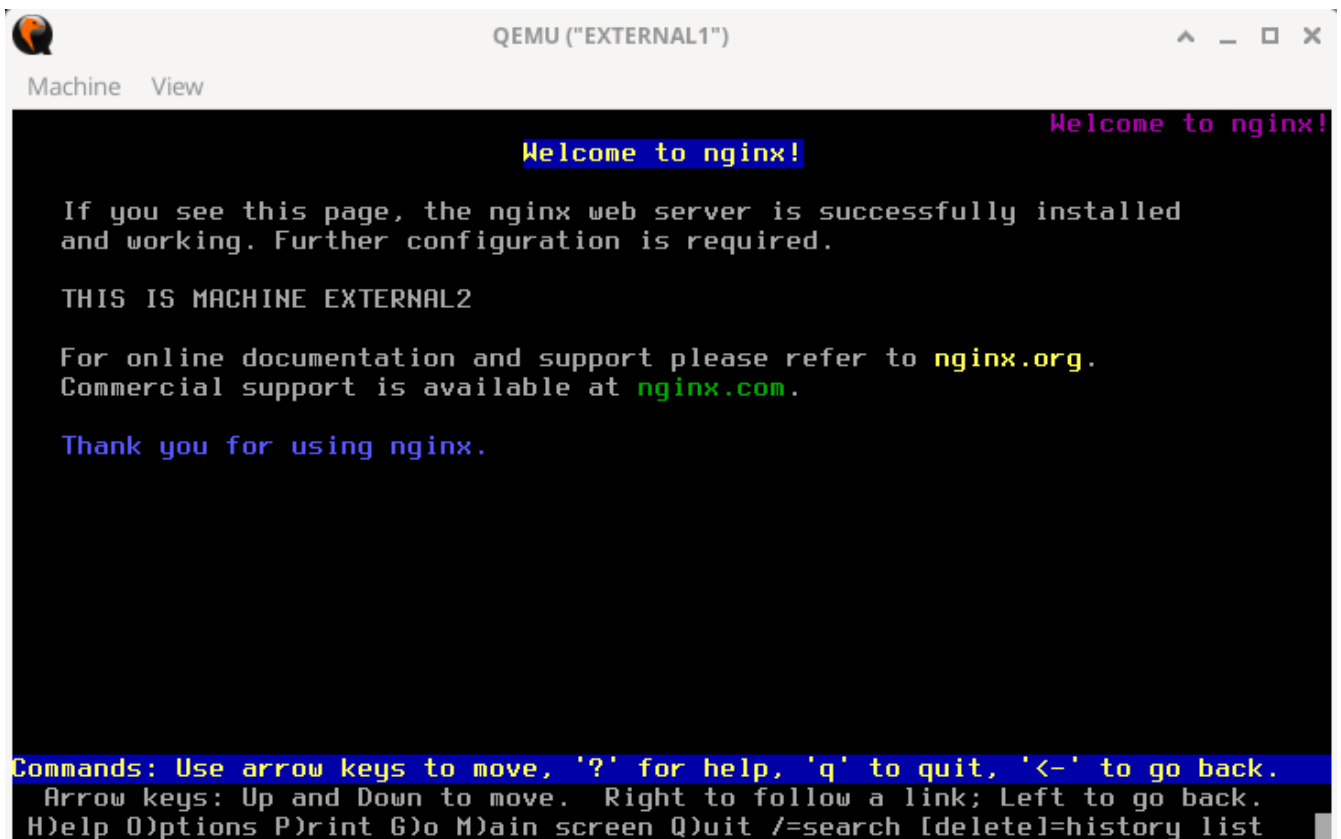
```
root@firewall2:~/bin # cat /var/log/security
Dec  2 15:10:25 firewall2 kernel: ipfw: 300 Eaction nat64lsn ICMPv6:128.0
[2001:db8:aaaa::c0a8:102] [2001:db8:bbbb::cb00:7114] in via em0
Dec  2 15:10:25 firewall2 kernel: ipfw: 400 Eaction nat64lsn ICMP:0.0 203.0.113.20
203.0.112.22 in via em1
Dec  2 15:10:26 firewall2 kernel: ipfw: 300 Eaction nat64lsn ICMPv6:128.0
[2001:db8:aaaa::c0a8:102] [2001:db8:bbbb::cb00:7114] in via em0
Dec  2 15:10:26 firewall2 kernel: ipfw: 400 Eaction nat64lsn ICMP:0.0 203.0.113.20
203.0.112.22 in via em1
Dec  2 15:10:29 firewall2 kernel: ipfw: 100 Accept ICMPv6:135.0 [2001:db8:12::1]
[2001:db8:12::2] out via em0
Dec  2 15:10:29 firewall2 kernel: ipfw: 100 Accept ICMPv6:136.0 [2001:db8:12::2]
[2001:db8:12::1] in via em0
Dec  2 15:10:30 firewall2 kernel: ipfw: 100 Accept ICMPv6:135.0 [2001:db8:12::2]
[2001:db8:12::1] in via em0
Dec  2 15:10:30 firewall2 kernel: ipfw: 100 Accept ICMPv6:136.0 [2001:db8:12::1]
```

```
[2001:db8:12::2] out via em0
```

Finally, a webpage request was made with:

```
# lynx external2.example.com
```

as shown below:



```
QEMU ("EXTERNAL1")
Machine View
Welcome to nginx!
Welcome to nginx!
If you see this page, the nginx web server is successfully installed
and working. Further configuration is required.
THIS IS MACHINE EXTERNAL2
For online documentation and support please refer to nginX.org.
Commercial support is available at nginX.com.
Thank you for using nginX.
Commands: Use arrow keys to move, '?' for help, 'q' to quit, '<-' to go back.
Arrow keys: Up and Down to move. Right to follow a link; Left to go back.
H)elp O)ptions P)rint G)o M)ain screen Q)uit /=search [delete]=history list
```

Figure 35. Retrieving the Webpage at `external2.example.com`



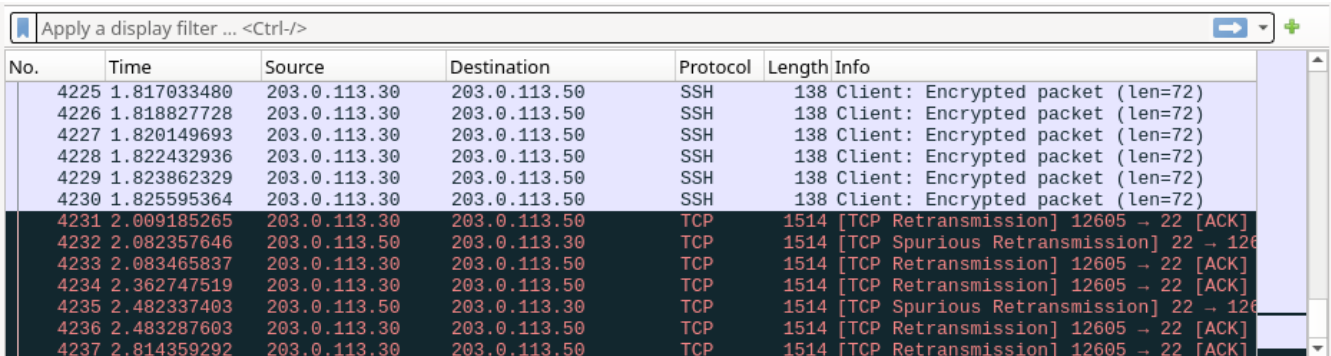
As noted in the Juniper documentation link above, "464XLAT only supports IPv4 in the client-server model, so it does not support IPv4 peer-to-peer communication or inbound IPv4 connections." Other technologies provide peer-to-peer communications.

Chapter 7. Other Keywords

This section covers some other lesser used keywords.

7.1. abort / abort6

The **abort** and **abort6** keywords interrupt the data stream between two endpoints. The effect of this keyword, is similar to the **reset** keyword, but there are important differences.



No.	Time	Source	Destination	Protocol	Length	Info
4225	1.817033480	203.0.113.30	203.0.113.50	SSH	138	Client: Encrypted packet (len=72)
4226	1.818827728	203.0.113.30	203.0.113.50	SSH	138	Client: Encrypted packet (len=72)
4227	1.820149693	203.0.113.30	203.0.113.50	SSH	138	Client: Encrypted packet (len=72)
4228	1.822432936	203.0.113.30	203.0.113.50	SSH	138	Client: Encrypted packet (len=72)
4229	1.823862329	203.0.113.30	203.0.113.50	SSH	138	Client: Encrypted packet (len=72)
4230	1.825595364	203.0.113.30	203.0.113.50	SSH	138	Client: Encrypted packet (len=72)
4231	2.009185265	203.0.113.30	203.0.113.50	TCP	1514	[TCP Retransmission] 12605 → 22 [ACK]
4232	2.082357646	203.0.113.50	203.0.113.30	TCP	1514	[TCP Spurious Retransmission] 22 → 12605 [ACK]
4233	2.083465837	203.0.113.30	203.0.113.50	TCP	1514	[TCP Retransmission] 12605 → 22 [ACK]
4234	2.362747519	203.0.113.30	203.0.113.50	TCP	1514	[TCP Retransmission] 12605 → 22 [ACK]
4235	2.482337403	203.0.113.50	203.0.113.30	TCP	1514	[TCP Spurious Retransmission] 22 → 12605 [ACK]
4236	2.483287603	203.0.113.30	203.0.113.50	TCP	1514	[TCP Retransmission] 12605 → 22 [ACK]
4237	2.814359292	203.0.113.30	203.0.113.50	TCP	1514	[TCP Retransmission] 12605 → 22 [ACK]

Figure 36. Abort and abort6 keywords

The above figure shows the effect of inserting the **firewall** rule:

```
# ipfw add 50 abort tcp from 203.0.113.30 to me
```

Unlike the **reset** keyword, there is no packet sent from the firewall to the source. What happens is that **ipfw** just starts dropping packets that match the rule. Since there are no more replies coming from the destination (here the firewall itself), the source endpoint issues retransmissions over and over. Eventually the source concludes that the connection is irrevocably broken and it closes the connection.



In the rule above, **all** TCP connections will be interrupted between the two systems.



In a TCP connection, **ipfw** will use dynamic rules if a **check-state** rule is already in place. If this is the case, issue the **abort** rule at a rule number **before** the check-state rule. Otherwise, it will have no effect.

7.2. mark / setmark

The **setmark** keyword functions similar to the **tag** keyword. If the packet matches the rule, **ipfw** applies a 32-bit identifier to the packet. This identifier (the "mark") is held with the packet internally inside **ipfw**. It is not sent with the packet on the wire and is not visible to any network

monitoring from tools like [tcpdump\(1\)](#) or [wireshark\(1\)](#).

Like **tags**, a **mark** can be used as another filtering device with other **ipfw** rules to do policy based routing or filtering. Note that only one mark can be applied at a time.

A big advantage of marks over tags are their ability to be matched as a lookup key in a table. Also, a mark can have a bitmask applied to it.

To explore **mark** and **setmark**, use the architecture of Simple NAT shown in [Simple NAT](#). Begin by creating the network with the **mkbr.sh** script and starting the VMs with the **runvm.sh** script shown in [Simple NAT](#).

```
# sudo /bin/sh mkbr.sh reset bridge0 tap1 tap4 bridge1 tap0 tap5
# /bin/sh runvm.sh firewall external1 internal
```

Assign the IP addresses as shown, and ensure all VMs have connectivity with adjacent systems.

On the **internal** VM, start up the **userv.sh** script with port number 5656. Then, on the **external1** VM, start up the **ucont.sh** server with the same port and a time value of 1 second. With no **ipfw** module loaded, the communications should succeed.



It may be necessary to examine the **ucont.sh** script and assign the correct address for the connection.

Before placing a **setmark** value on a packet, load the **ipfw** module and redirect the output of the **ipfw** log by setting the `sysctl` to log to `syslog`:

```
# kldload ipfw
# sysctl net.inet.ip.fw.verbose=1
```

Now insert the following firewall rules and examine the log file `/var/log/security`:

```
# ipfw add 1000 allow log udp from any to 10.10.10.20 dst-port 5656
01000 allow log udp from any to 10.10.10.20 5656
#
# tail -f /var/log/security
Dec 29 22:32:33 firewall kernel: ipfw: 1000 Accept UDP 203.0.113.10:30463
10.10.10.20:5656 in via em1
Dec 29 22:32:33 firewall kernel: ipfw: 1000 Accept UDP 203.0.113.10:30463
10.10.10.20:5656 out via em0
Dec 29 22:32:36 firewall kernel: ipfw: 1000 Accept UDP 203.0.113.10:24588
10.10.10.20:5656 in via em1
```

```
Dec 29 22:32:36 firewall kernel: ipfw: 1000 Accept UDP 203.0.113.10:24588
10.10.10.20:5656 out via em0
```

Now add the following rule to apply the **mark** value of 20 (decimal) and observe the change in the logs:

```
# ipfw add 500 setmark 20 log udp from any to 10.10.10.20 dst-port 5656
00500 setmark 0x14 log udp from any to 10.10.10.20 5656
#
# tail -f /var/log/security
Dec 29 22:41:20 firewall kernel: ipfw: 1000 Accept UDP 203.0.113.10:27955
10.10.10.20:5656 in via em1
Dec 29 22:41:20 firewall kernel: ipfw: 1000 Accept UDP 203.0.113.10:27955
10.10.10.20:5656 out via em0
Dec 29 22:41:23 firewall kernel: ipfw: 1000 Accept UDP 203.0.113.10:37423
10.10.10.20:5656 in via em1
Dec 29 22:41:23 firewall kernel: ipfw: 1000 Accept UDP 203.0.113.10:37423
10.10.10.20:5656 out via em0
Dec 29 22:41:25 firewall kernel: ipfw: 500 SetMark 0x14 UDP 203.0.113.10:45176
10.10.10.20:5656 in via em1
Dec 29 22:41:25 firewall kernel: ipfw: 1000 Accept UDP 203.0.113.10:45176
10.10.10.20:5656 mark:0x14 in via em1
Dec 29 22:41:25 firewall kernel: ipfw: 500 SetMark 0x14 UDP 203.0.113.10:45176
10.10.10.20:5656 out via em0
Dec 29 22:41:25 firewall kernel: ipfw: 1000 Accept UDP 203.0.113.10:45176
10.10.10.20:5656 mark:0x14 out via em0
Dec 29 22:41:27 firewall kernel: ipfw: 500 SetMark 0x14 UDP 203.0.113.10:21444
10.10.10.20:5656 in via em1
Dec 29 22:41:27 firewall kernel: ipfw: 1000 Accept UDP 203.0.113.10:21444
10.10.10.20:5656 mark:0x14 in via em1
```

7.3. NPTv6

IPv6-to-IPv6 Network Prefix Translation (NPTv6) is the process of translating IPv6 header source and destination addresses. Functionally, it is similar to the more well understood Network Address Translation, but without the need to maintain state. It is only the IPv6 source and destination addresses that are translated. The idea here is to allow an edge network to have its own independent addressing scheme while being able to exchange IPv6 traffic with external IPv6 networks through the use of an NPTv6 Translator

[RFC 6296](#) is the definitive document on NPTv6. The example in this section is taken from Sections 2.1 of that document.

The architecture for these examples is based on Simple NAT as in the previous section.

7.3.1. NPTv6 Setup

Use the setup instructions shown in [Simple NAT](#) but use the IPv6 addressing as shown below:

NPTv6: The Simplest Case

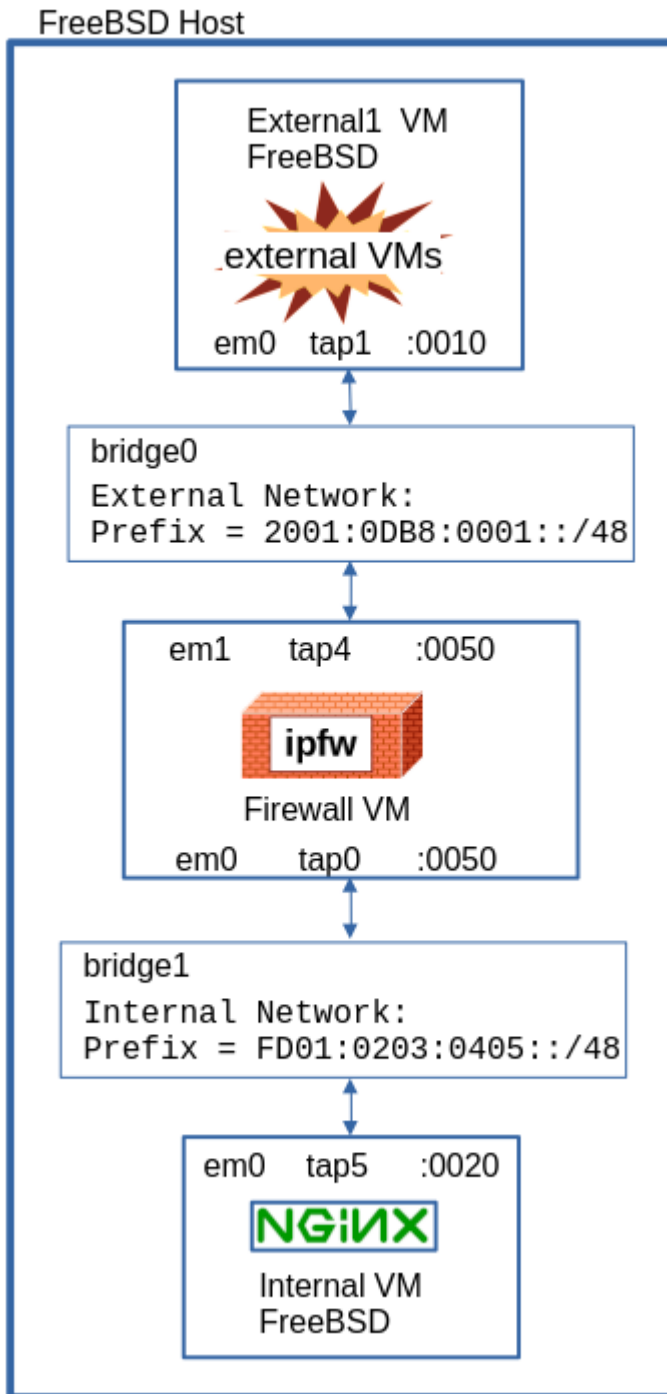


Figure 37. NPTv6 Simple Case

At first glance, this appears to be a simple IPv6 forwarding example. However in this case, **NPTv6** changes the actual packet source and destination addresses, so no forwarding is needed.

[ipfw\(8\)](#) explains the syntax of the NPTv6 command and options, but there are a number of details that need to be set up correctly. Use the following as a guide:

On the FreeBSD host:

```
$ sudo /bin/sh mkbr.sh reset bridge0 tap1 tap4 bridge1 tap0 tap5
$ /bin/sh runvm.sh external1 firewall internal
```

Ensure all IPv6 addresses on all VMs are set up correctly.

On the firewall VM:

```
# kldunload ipfw_nptv6
# kldunload ipfw

# kldload ipfw
# kldload ipfw_nptv6

# sysctl net.inet.ip.fw.one_pass=0
# sysctl net.inet.ip.fw.verbose=1

# ipfw -q flush

# Set up the NPTv6 instance.
# ipfw nptv6 foo create int_prefix fd01:0203:0405:: ext_prefix 2001:0db8:0001::
prefixlen 48

# Rules for nptv6
# ipfw add 500 allow log ipv6-icmp from any to any icmp6types 135,136 // allow
neighbor solicitation
# *ipfw add 2000 nptv6 foo log ip6 from fd01:0203:0405::/48 to any
# ipfw add 3000 allow ip6 from any to any
```

As noted in [ipfw\(8\)](#), the `sysctl net.inet6.ip6.forwarding=1` must be applied or NPTv6 will silently stop working.

7.3.2. NPTv6 Testing

Set up a UDP listener on the **external1** VM. Using the **userv.sh** (and its **ucon.sh** partner) is possible, but that would require editing the scripts to set up an IPv6 address. Try this method instead:

```
On the external1 VM:
# Listen for a UDP packet
$ ncat -l -k -u -6 2001:0db8:0001::10 5656

On the internal VM:
# Set up the default route for IPv6
# route -6 add default fd01:0203:0405::0050
#
# Send the desired UDP packet.
$ echo "testing123" | ncat -6 -u 2001:0db8:0001::10 5656
```

In the setup section above, logging to `syslogd` was set up, so the results can be seen by examining the tail end of `/var/log/security`:

```
Dec 31 19:51:44 firewall kernel: ipfw: 2000 Eaction nptv6 UDP [fd01:203:405::20]:52451
[2001:db8:1::10]:5656 in via em0
```

The output of a `tcpdump` on `external1` shows:

```
root@external1:~ # tcpdump -n -i em0 -X "udp"
tcpdump: verbose output suppressed, use -v[v]... for full protocol decode
listening on em0, link-type EN10MB (Ethernet), snapshot length 262144 bytes
19:51:43.827543 IP6 2001:db8:1:d54f::20.52451 > 2001:db8:1::10.5656: UDP, length 11
    0x0000:  600a 145a 0013 113f 2001 0db8 0001 d54f  `..Z...?.....0
    0x0010:  0000 0000 0000 0020 2001 0db8 0001 0000  .....
    0x0020:  0000 0000 0000 0010 cce3 1618 0013 f72b  .....+
    0x0030:  7465 7374 696e 6731 3233 0a          testing123.
```

The highlighted section shows the effect of the NPTv6 translation. (See [RFC 6296](#), Section 3, for details.)

7.4. ipttl

The `ipttl` (Time to Live or TTL) keyword identifies packets that have specific TTL characteristics. [ipfw\(8\)](#) notes that the `ipttl` keyword will accept a single value, a list of values, or a range of values, in the same syntax as that used for the `ports` keyword. (Recall the discussion of lists and ranges in the [Notes on Rule Numbering](#).)

`ipttl` is one of a number of `ipfw` keywords that work on individual fields of packets flowing through the firewall. Similar keywords include `ipid`, `iplen`, `ipprecedence`, etc. The `ipttl` keyword controls the lifetime of the packet on the network (see below).

7.4.1. ipttl Setup

Use the setup instructions shown in [Simple NAT](#) with IPv4 addressing, not IPv6.

Also, this example will use the `hping3` command. (To download the `hping3` package, reset the `internal` VM for access to the Internet, and download the package with `pkg install hping3`. Remember to reset for Simple NAT IP addressing for this example.)

Refer to [hping3\(8\)](#) for details.

On the **external1** VM:

```
# Listen for a UDP packet
# ncat -l -k -u 203.0.113.10 5656
```

On the **internal** VM:

```
# Send the desired UDP packet.
Now, deliberately set the initial TTL to 13.
# hping3 --sign "test for ttl 13" --count 1 --udp --ttl 13 --destport 5656
203.0.113.10
```

Without **ipfw** in place, the result should be similar to:

```
root@external1:~ # tcpdump -n -i em0 -X -vv "udp"
tcpdump: listening on em0, link-type EN10MB (Ethernet), snapshot length 262144 bytes
14:49:16.658160 IP (tos 0x0, ttl 12, id 6642, offset 0, flags [none], proto UDP (17),
length 43)
  10.10.10.20.2472 > 203.0.113.10.5656: [udp sum ok] UDP, length 15
    0x0000:  4500 002b 19f2 0000 0c11 44a8 0a0a 0a14  E..+.....D.....
    0x0010:  cb00 710a 09a8 1618 0017 3013 7465 7374  ..q.....0.test
    0x0020:  2066 6f72 2074 746c 2031 3300 0000      .for.ttl.13...
```

The IP Time to Live option was set up to prevent IP packets from bouncing around the Internet forever. [RFC 791](#) initially intended that the value would be considered an actual time value (number of seconds) and that each module processing the packet would subtract processing time from the initial value. This was later changed to an integer identifying a "hop count" where the initial value (now 64) is decremented by every router or gateway or forwarding device, such as a firewall.

In this case the **firewall** VM, even though it is not running firewall software, is still a 'forwarding device' and decrements the count as it forwards the packet.

7.4.2. ipttl Testing

To examine the **ipttl** keyword follow this example:

```
# kldload ipfw
# sysctl net.inet.ip.fw.verbose=1

# Count all packets as the flow through
# ipfw add 800 count ip from any to any

# Count all packets with TTL of exactly 13 as they enter em0.
# ipfw add 900 count ip from any to any ipttl 13
```

```
# Allow and log packets with TTL of exactly 13 as they enter em0.
# ipfw add 1000 allow log udp from any to any ipttl 13

# Just before the packet exits, the IP stack decrements the ttl,
# so the following rule is also needed for the packet to exit out em1.
# ipfw add 1050 allow log udp from any to any ipttl 12

# Count any other ip packets after the ipttl rule
# ipfw add 1100 count ip from any to any
```

Below is a sample run of **ncat** and **hping3** commands to test the above rules:

```
# echo "UDP with default TTL" | ncat -u 203.0.113.10 5656
# echo "UDP with default TTL" | ncat -u 203.0.113.10 5656

# hping3 --sign "UDP with TTL=13" --count 1 --udp --ttl 13 --destport 5656
203.0.113.10
# hping3 --sign "UDP with TTL=13" --count 1 --udp --ttl 13 --destport 5656
203.0.113.10
```

The results show the first two packets with default TTL values (64) were not passed by the firewall. The third and fourth packets were passed. The traces below show the input packet in interface **em0** with ttl 13, and the output packet on **em1** with ttl 12.

```
# tcpdump -n -i em0 -X -vvv "udp"
tcpdump: listening on em0, link-type EN10MB (Ethernet), snapshot length 262144 bytes
03:26:42.875634 IP (tos 0x0, ttl 13, id 17385, offset 0, flags [none], proto UDP (17),
length 43)
  10.10.10.20.1648 > 203.0.113.10.5656: [udp sum ok] UDP, length 15
    0x0000:  4500 002b 43e9 0000 0c11 1ab1 0a0a 0a14  E..+C.....
    0x0010:  cb00 710a 0670 1618 0017 1d07 5544 5020  ..q..p.....UDP.
    0x0020:  7769 7468 2054 544c 3d31 3300 0000          with.TTL=13...

#
#
# tcpdump -n -i em1 -X -vvv "udp"
tcpdump: listening on em1, link-type EN10MB (Ethernet), snapshot length 262144 bytes
03:27:37.903863 IP (tos 0x0, ttl 12, id 33936, offset 0, flags [none], proto UDP (17),
length 43)
  10.10.10.20.2539 > 203.0.113.10.5656: [udp sum ok] UDP, length 15
    0x0000:  4500 002b 8490 0000 0c11 da09 0a0a 0a14  E..+.....
    0x0010:  cb00 710a 09eb 1618 0017 198c 5544 5020  ..q.....UDP.
    0x0020:  7769 7468 2054 544c 3d31 3300 0000          with.TTL=13...
```


7.5. tcpdataalen

The **tcpdataalen** keyword is one of several related keywords:

- **tcpack, tcpdataalen, tcpflags, tcpmss, tcpseq, tcpwin, tcptoptions**

These keywords are not often used.

However, there is one very important use case. From time to time, an Internet worm - a malicious packet that gets resent to all local and remote hosts matching some criteria - makes its way onto the Internet. Quick thinking network security administrators can sometimes identify a unique characteristic of these malicious packets such as all packets having the same length - akin to **tcpdataalen**, or a certain set of **tcptoptions**.

In this example, the **firewall** VM is running the **tserve.sh 5656** script.



It may be necessary to edit the **tserve.sh** script to listen on the correct interface (em1) for this example.

The example below, using the [Simple NAT](#) setup and addressing, configures **ipfw** to deny all packets having a TCP data length of a certain value range. But, it also allows the completion of the TCP 3-way handshake. When the handshake is completed, one of these ranges will cause the malicious packet to be denied. Keep in mind, this is the length of the TCP data payload, not the overall length of the packet.

```
# ipfw -q flush
# ipfw add 10 deny tcp from any to me tcpdataalen 10-19
# ipfw add 20 deny tcp from any to me tcpdataalen 20-29
# ipfw add 30 deny tcp from any to me tcpdataalen 30-39
# ipfw add 40 deny tcp from any to me tcpdataalen 40-49
# ipfw add 50 deny tcp from any to me tcpdataalen 50-59
# ipfw add 60 deny tcp from any to me tcpdataalen 60-69
# ipfw add 70 deny tcp from any to me tcpdataalen 70-79
# ipfw add 80 deny tcp from any to me tcpdataalen 80-89
# ipfw add 90 deny tcp from any to me tcpdataalen 90-99
# ipfw add 500 check-state
# ipfw add 1000 allow tcp from any to any 5656 setup keep-state
#
# ipfw show
00010 0 0 deny log tcp from any to me tcpdataalen 10-19
00020 0 0 deny log tcp from any to me tcpdataalen 20-29
00030 0 0 deny log tcp from any to me tcpdataalen 30-39
00040 0 0 deny log tcp from any to me tcpdataalen 40-49
00050 0 0 deny log tcp from any to me tcpdataalen 50-59
00060 0 0 deny log tcp from any to me tcpdataalen 60-69
00070 0 0 deny log tcp from any to me tcpdataalen 70-79
00080 0 0 deny log tcp from any to me tcpdataalen 80-89
00090 0 0 deny log tcp from any to me tcpdataalen 90-99
```

```
00500 0 0 check-state :default
01000 0 0 allow log tcp from any to any setup keep-state :default
65535 0 0 deny ip from any to any
```

And a test using **ncat** directly from **external1**:

```
# echo "1234567890123456789012345678901234" | ncat 203.0.113.50 5656
```

The TCP 3-way handshake completes, but the packet containing the data payload is stopped by rule 30 as shown below:

```
# ipfw show
00010 0 0 deny log tcp from any to me tcpdatalen 10-19
00020 0 0 deny log tcp from any to me tcpdatalen 20-29
00030 13 1066 deny log tcp from any to me tcpdatalen 30-39
00040 0 0 deny log tcp from any to me tcpdatalen 40-49
00050 0 0 deny log tcp from any to me tcpdatalen 50-59
00060 0 0 deny log tcp from any to me tcpdatalen 60-69
00070 0 0 deny log tcp from any to me tcpdatalen 70-79
00080 0 0 deny log tcp from any to me tcpdatalen 80-89
00090 0 0 deny log tcp from any to me tcpdatalen 90-99
00500 0 0 check-state :default
01000 8 420 allow log tcp from any to any 5656 setup keep-state :default
65535 0 0 deny ip from any to any
```

The reason for the excessive number of packets denied is TCP retransmission trying to account for the dropped packet as shown in the [wireshark\(1\)](#) trace below.

No.	Time	Source	Destination	Protocol	Length	Info
3	0.001379247	203.0.113.30	203.0.113.50	TCP	74	49036 → 5656 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=64 SACK_PERM TSval=3998399885 TSecr=0
4	0.002027832	203.0.113.50	203.0.113.30	TCP	74	5656 → 49036 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1460 WS=64 SACK_PERM TSval=3849425364 TSecr=3998399885
5	0.002587561	203.0.113.30	203.0.113.50	TCP	66	49036 → 5656 [ACK] Seq=1 Ack=1 Win=65728 Len=0 TSval=3998399885 TSecr=3849425364
6	0.003985120	203.0.113.30	203.0.113.50	TCP	97	49036 → 5656 [PSH, ACK] Seq=1 Ack=1 Win=65728 Len=31 TSval=3998399885 TSecr=3849425364
7	0.004954317	203.0.113.30	203.0.113.50	TCP	66	49036 → 5656 [FIN, ACK] Seq=32 Ack=1 Win=65728 Len=0 TSval=3998399885 TSecr=3849425364
8	0.005726984	203.0.113.50	203.0.113.30	TCP	66	[TCP Window Update] 5656 → 49036 [ACK] Seq=1 Ack=1 Win=65728 Len=0 TSval=3849425369 TSecr=3998399885
9	0.240093167	203.0.113.30	203.0.113.50	TCP	97	[TCP Retransmission] 49036 → 5656 [FIN, PSH, ACK] Seq=1 Ack=1 Win=65728 Len=31 TSval=3998400129 TSecr=3849425369
10	0.512625959	203.0.113.30	203.0.113.50	TCP	97	[TCP Retransmission] 49036 → 5656 [FIN, PSH, ACK] Seq=1 Ack=1 Win=65728 Len=31 TSval=3998400461 TSecr=3849425369
11	0.846579720	203.0.113.30	203.0.113.50	TCP	97	[TCP Retransmission] 49036 → 5656 [FIN, PSH, ACK] Seq=1 Ack=1 Win=65728 Len=31 TSval=3998400735 TSecr=3849425369
12	1.312436531	203.0.113.30	203.0.113.50	TCP	97	[TCP Retransmission] 49036 → 5656 [FIN, PSH, ACK] Seq=1 Ack=1 Win=65728 Len=31 TSval=3998401201 TSecr=3849425369
13	2.013486215	203.0.113.30	203.0.113.50	TCP	97	[TCP Retransmission] 49036 → 5656 [FIN, PSH, ACK] Seq=1 Ack=1 Win=65728 Len=31 TSval=3998401901 TSecr=3849425369
14	3.212502112	203.0.113.30	203.0.113.50	TCP	97	[TCP Retransmission] 49036 → 5656 [FIN, PSH, ACK] Seq=1 Ack=1 Win=65728 Len=31 TSval=3998403101 TSecr=3849425369
15	5.303647754	203.0.113.30	203.0.113.50	TCP	97	[TCP Retransmission] 49036 → 5656 [FIN, PSH, ACK] Seq=1 Ack=1 Win=65728 Len=31 TSval=3998405252 TSecr=3849425369
16	9.412525237	203.0.113.30	203.0.113.50	TCP	97	[TCP Retransmission] 49036 → 5656 [FIN, PSH, ACK] Seq=1 Ack=1 Win=65728 Len=31 TSval=3998409301 TSecr=3849425369
17	17.312439263	203.0.113.30	203.0.113.50	TCP	97	[TCP Retransmission] 49036 → 5656 [FIN, PSH, ACK] Seq=1 Ack=1 Win=65728 Len=31 TSval=3998417201 TSecr=3849425369
18	32.913152680	203.0.113.30	203.0.113.50	TCP	97	[TCP Retransmission] 49036 → 5656 [FIN, PSH, ACK] Seq=1 Ack=1 Win=65728 Len=31 TSval=3998432801 TSecr=3849425369
19	48.512817048	203.0.113.30	203.0.113.50	TCP	97	[TCP Retransmission] 49036 → 5656 [FIN, PSH, ACK] Seq=1 Ack=1 Win=65728 Len=31 TSval=3998448401 TSecr=3849425369
20	64.112307687	203.0.113.30	203.0.113.50	TCP	97	[TCP Retransmission] 49036 → 5656 [FIN, PSH, ACK] Seq=1 Ack=1 Win=65728 Len=31 TSval=3998464001 TSecr=3849425369
21	70.112439002	203.0.113.30	203.0.113.50	TCP	66	49036 → 5656 [RST, ACK] Seq=33 Ack=1 Win=0 Len=0 TSval=3998479601 TSecr=3849425369
22	79.713163922	203.0.113.50	203.0.113.30	TCP	66	[TCP Window Update] 5656 → 49036 [ACK] Seq=1 Ack=1 Win=65604 Len=0 TSval=3849565077 TSecr=3998399885
23	79.713703367	203.0.113.30	203.0.113.50	TCP	54	49036 → 5656 [RST] Seq=1 Win=0 Len=0

Figure 38. Denying Packet Based on TCP Data Length

Eventually TCP gives up and shuts down the connection.

7.6. verrevpath / versrcreach / antispoof

These keywords all work to determine if an incoming packet is legitimate.

As noted in [ipfw\(8\)](#), **verrevpath** ("verify reverse path") looks up the incoming packet's source address in the routing table.

Quoting: *"If the interface on which the packet entered the system matches the outgoing interface for the route, the packet matches. If the interfaces do not match up, the packet does not match. All outgoing packets or packets with no incoming interface match."*

Setting up on the FreeBSD host:

```
% cd ~/ipfw-primer/ipfw/HOST_SCRIPTS
% sudo /bin/sh mkbr.sh reset bridge0 tap0 tap5 bridge1 tap4 tap1
% /bin/sh runvm.sh external1 firewall internal
```

Consider the figure below (same as Simple NAT):

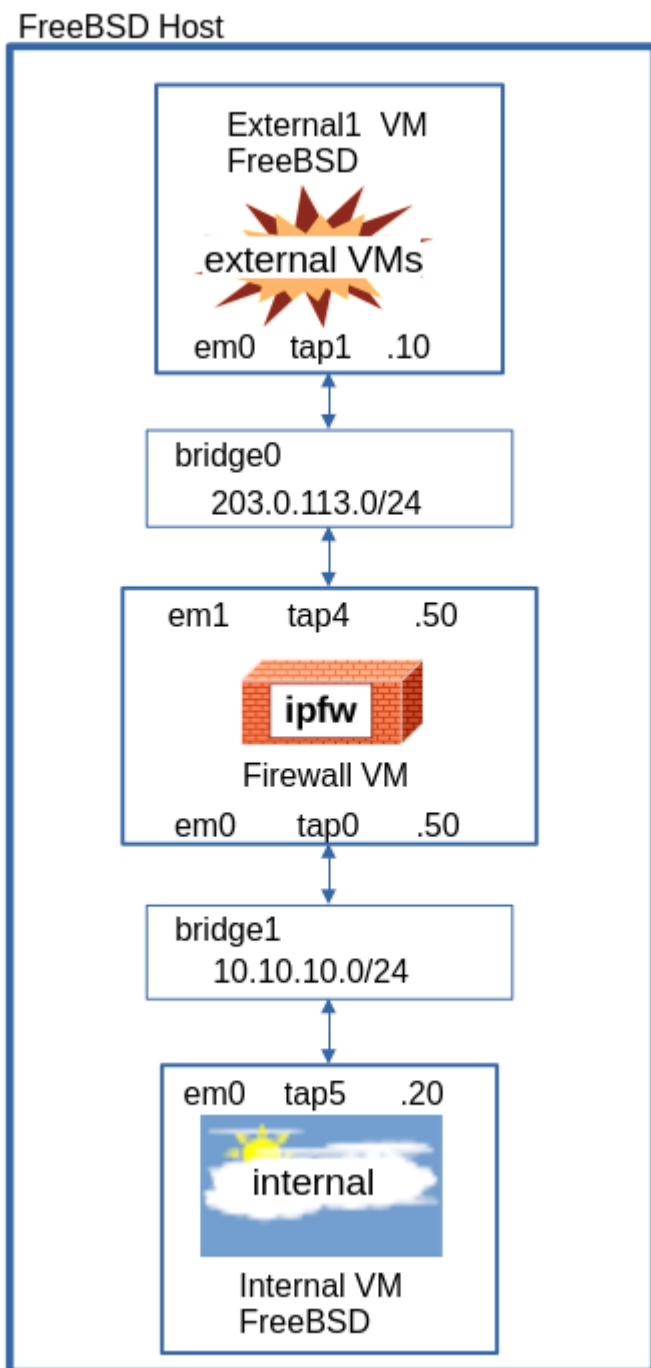


Figure 39. verrevpath Example

In this figure, the firewall has interface **em0** directly connected to the **10.10.10.0/24** network and the **em1** interface directly connected to the **203.0.113.0/24** network.

The **firewall** VM interfaces and routing table are shown in the text below:

```

root@firewall:~ # ifconfig em0
em0: flags=1008843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST,LOWER_UP> metric 0 mtu 1500
options=48525bb<RXCSUM, TXCSUM, VLAN_MTU, VLAN_HWTAGGING, JUMBO_MTU, VLAN_HWCSUM, TS04, LRO, W
OL_MAGIC, VLAN_HWFILTER, VLAN_HWTSO, HWSTATS, MEXTPG>
ether 02:49:50:46:57:41
inet 10.10.10.50 netmask 0xffffffff broadcast 10.10.10.255
media: Ethernet autoselect (1000baseT <full-duplex>)
  
```

```

status: active
nd6 options=29<PERFORMNUD,IFDISABLED,AUTO_LINKLOCAL>
root@firewall:~ #
root@firewall:~ # ifconfig em1
em1: flags=1008843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST,LOWER_UP> metric 0 mtu 1500

options=48525bb<RXCSUM,TXCSUM,VLAN_MTU,VLAN_HWTAGGING,JUMBO_MTU,VLAN_HWCSUM,TSO4,LRO,W
OL_MAGIC,VLAN_HWFILTER,VLAN_HWTSO,HWSTATS,MEXTPG>
ether 02:49:50:46:57:42
inet 203.0.113.50 netmask 0xfffff00 broadcast 203.0.113.255
media: Ethernet autoselect (1000baseT <full-duplex>)
status: active
nd6 options=29<PERFORMNUD,IFDISABLED,AUTO_LINKLOCAL>
root@firewall:~ #
root@firewall:~ # netstat -rn
Routing tables

Internet:
Destination      Gateway          Flags           Netif Expire
10.10.10.0/24    link#1          U               em0
10.10.10.50     link#3          UHS             lo0
127.0.0.1       link#3          UH              lo0
203.0.113.0/24  link#2          U               em1
203.0.113.50   link#3          UHS             lo0

Internet6:
Destination      Gateway          Flags           Netif Expire
::/96            link#3          URS             lo0
::1              link#3          UHS             lo0
::ffff:0.0.0.0/96 link#3          URS             lo0
fe80::%lo0/10   link#3          URS             lo0
fe80::%lo0/64   link#3          U               lo0
fe80::1%lo0     link#3          UHS             lo0
ff02::/16       link#3          URS             lo0
root@firewall:~ #

```

If a packet came in on the **em0** interface with a source address that was not in the **10.10.10.0/24** network, the above quote says the packet should be dropped.

The following example tests this with the **ncat** program which has an option to set the source IP.

First, set up **ipfw** on the **firewall** VM to allow any UDP packets as shown.

Then, set up the **firewall** VM to run **sh userv.sh 5656**, the service to receive UDP packets on the identified port. Next, send one packet from the **internal** VM with **echo "hello from internal VM" | ncat -u 10.10.10.50 5656**.

```

root@firewall:~/bin # ipfw add 1000 allow udp from any to me verrevpath
01000 allow udp from any to me verrevpath
root@firewall:~/bin #

```

```

root@firewall:~/bin # ipfw show
01000 0 0 allow udp from any to me verrevpath
65535 0 0 deny ip from any to any
root@firewall:~/bin #
root@firewall:~/bin # sh userv.sh 5656
PORT1 = [5656]
Starting UDP listener on [10.10.10.50],[5656]
hello from internal VM
^Croot@firewall:~/bin #
root@firewall:~/bin # ipfw show
01000 1 51 allow udp from any to me verrevpath
65535 0 0 deny ip from any to any
root@firewall:~/bin #

```

So far, so good. This is expected behavior.

Now zero the rule counts on the **firewall** VM and send a similar message from the **internal** VM, but this time spoof the source address. This requires adding an alias IP address to the interface on the **internal** VM:

```

root@internal:~/bin # ifconfig em0 4.4.4.4/32 alias
root@internal:~/bin #
root@internal:~/bin # echo "hello 2 from internal VM" | ncat -u -s 4.4.4.4 10.10.10.50
5656
root@internal:~/bin #

```

Now, rule 1000 prevents the matching of the incoming packet with a spoofed source address and no packet is received by the **userv.sh** service. Instead, the packet is handled by the default deny rule:

```

root@firewall:~/bin # sh userv.sh 5656
PORT1 = [5656]
Starting UDP listener on [10.10.10.50],[5656]
^Croot@firewall:~/bin #
root@firewall:~/bin #
root@firewall:~/bin # ipfw show
01000 0 0 allow udp from any to me verrevpath
65535 1 53 deny ip from any to any
root@firewall:~/bin #

```

The other keywords in this section, **verscreach** and **antispoof** operate in a similar manner. Check the man page for the slight differences between them.

7.7. jail

Jails are an important component of FreeBSD and have been a part of the base system since FreeBSD 4. **ipfw** works in tandem with jails to provide networking security. As discussed in the [FreeBSD Handbook Section on Jails and Networking](#), there are three types of jail networking setups. This section discusses the first two:

- Host Networking Setup
- Virtual Networking (VNET) Setup

7.7.1. Host-based Jail Networking

In this type of networking setup, the jail shares the host networking stack. The jail has the same IP address and interface as the host.

Recall that the **jail1** VM has different characteristics than the standard VMs used in this book. It has 8GB memory, a bigger disk, and is running ZFS for its filesystem.

Instructions for setting up this type of jail are found in the FreeBSD Handbook Section [Creating a Thin Jail Using OpenZFS Snapshots](#).

```
% cd ~/ipfw-primer/ipfw/HOST_SCRIPTS
% sudo /bin/sh mkbr.sh reset bridge0 tap12 host_interface
% /bin/sh runvm.sh jail1
```

Set up the **jail1** VM to use DHCP addressing and follow the instructions in the handbook to create a **thinjail** using ZFS, including creating `/etc/jail.conf` with the parameters shown in that section.

Once that is completed, reconfigure the FreeBSD host for this example.

Set up the **external1** and **jail1** VMs with these commands on the FreeBSD host:

```
% cd ~/ipfw-primer/ipfw/HOST_SCRIPTS
% sudo /bin/sh mkbr.sh reset bridge0 tap1 tap12
% /bin/sh runvm.sh external1 jail1
```

and use the addressing shown in the figure below.

FreeBSD Host

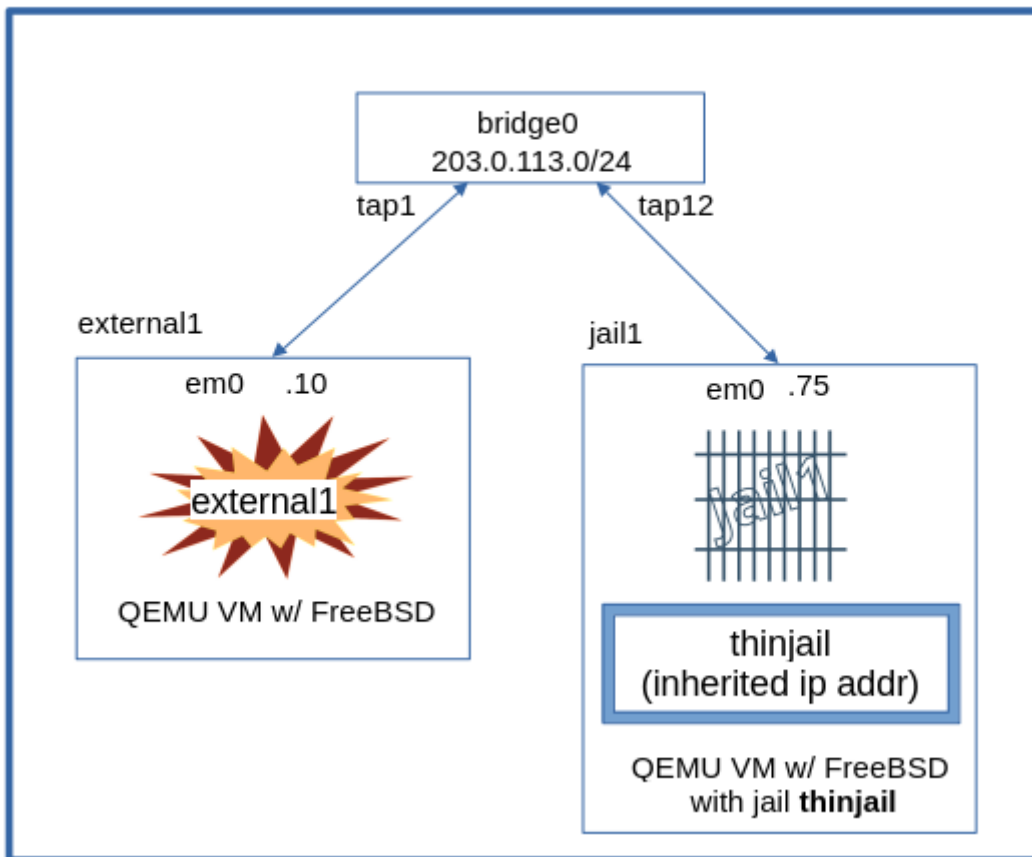


Figure 40. Jail With Host Based Networking

The typical jail configuration file for this setup uses the following network configuration:

```
jailname {  
    . . .  
    # Network  
    ip4 = inherit;  
    interface = em0;  
    . . .  
}
```

Login to the `jail1` VM and start the jail with:

```
# service jail onestart thinjail
```

Access the jail with the `jexec` command:

```
# jexec -u root thinjail
```

The jail named **thinjail** is now using the host `/etc/jail.conf` example, which uses the inherited IPv4 network stack, for the jail.



There are now *three* different command line environments - the FreeBSD **host**, the QEMU **jail1** VM, and the **thinjail** running inside the **jail1** VM. Keep track of which

command line you are using by watching the shell prompt.

Here, it is the host that controls the network stack and all **ipfw** commands (loading, unloading, adding/deleting rules, etc.) must be done from the host. The jail root user does not have permission to operate **ipfw** inside the jail.

```
root@jail1:~ # jexec -u root thinjail
root@thinjail:/ #
root@thinjail:/ # kldstat
Id Refs Address          Size Name
 1   18 0xffffffff80200000    1f370e8 kernel
 2    1 0xffffffff82138000     77d8 cryptodev.ko
 3    1 0xffffffff82140000    5cd608 zfs.ko
 4    1 0xffffffff83018000     3220 intpm.ko
 5    1 0xffffffff8301c000     2178 smbus.ko
root@thinjail:/ #
root@thinjail:/ # kldload ipfw
kldload: can't load ipfw: Operation not permitted
root@thinjail:/ #
```

Figure 41. Jail With Host Based Networking

All **ipfw** configuration for the jail must be done on the host. **ipfw** provides the **jail** keyword for this purpose. For IP communications, this keyword applies primarily to **outbound** packets from the jail. Inbound packets to the jail, follow the normal host rules.

By default, if the **thinjail** runs **nc -l 203.0.113.75 5656**, it opens up a TCP socket listening on port 5656 in the **jail1** VM. If instead, the **jail1** VM runs the identical command in the **jail1** VM, the listening socket is not visible to the **thinjail**.

The conditions for outside access to **thinjail** rely on the host network, and the **jail jailname** keyword is not needed.

```
root@jail1:# kldload ipfw
ipfw2 (+ipv6) initialized, divert loadable, nat loadable, default to deny, logging
disabled
root@jail1:#
root@jail1:# ipfw add 100 check-state
root@jail1:# ipfw add 1000 allow tcp from any to me dst-port 5656 setup keep-state
```

This rule on the host system will allow a connection from the **external1** VM to reach the above **nc -l 203.0.113.75 5656** running inside **thinjail**.

For outbound TCP communication, rule 2000 below, using the **jail thinjail** keyword is required. For the most part, the **ipfw** rules used elsewhere in this book are applicable here with the addition of the **jail jailname** keyword.

```
# ipfw add 100 check-state
# ipfw add 1000 allow tcp from any to me dst-port 5656 setup keep-state
# ipfw add 2000 allow tcp from me to any setup keep-state jail thinjail
```



Always provide the jail *name* rather than a numeric ID. If the jail is restarted for any reason it may get a new jail ID number and an existing rule with a jail number will be immediately out of date. The rule will have to be re-entered using the jail name.



When entering a rule with a jail name, **ipfw** will lookup the name and reply with the number. So even when listing or showing the ruleset, **ipfw** will always show the number not the name. Use the `jls` command to show the jail ID name and number.



It is a good idea to compartmentalize the rules for each jail in a file with the jail name. That way, if a jail is restarted, the specific file can be rerun to update the **ipfw** rules on the host.

7.7.2. Virtual Network (VNET) Jail Networking

A more advanced setup is using the VNET networking capabilities of FreeBSD for the jail. There are many good online tutorials on setting up VNET jails. This section is focused on the use of **ipfw** with a VNET network for the jail.

The architecture for this setup is shown in the figure below and is similar to that used in the last section.

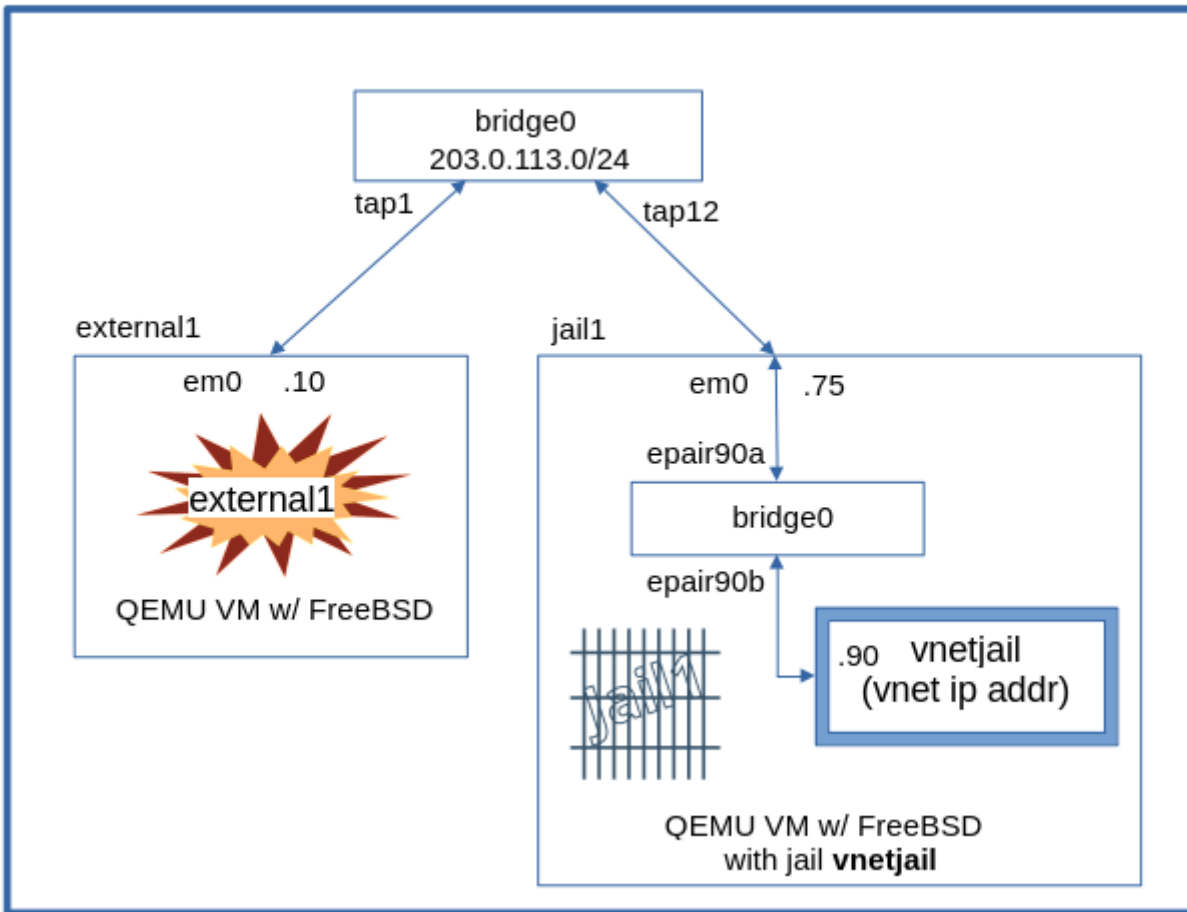


Figure 42. Jail With VNET Based Networking



While there are two `bridge0` interfaces shown in the diagram, they are completely unrelated. The top `bridge0` resides on the FreeBSD host and connects the `external1` and `jail1` VMs. The bottom `bridge0` resides *inside* the `jail1` VM and connects the `jail1` `em0` interface with the `epair(4)` interface attached to the `vnetjail` jail.

For this section, create a second thinjail named `vnetjail` as follows:

```
root@jail1:# zfs clone zroot/jails/templates/14.2-RELEASE@base
zroot/jails/containers/vnetjail
```

See below for configuring `/etc/jail.conf`.



Configuring multiple jails can be done with separate sections in `/etc/jail.conf`, or by creating separate configuration files in `/etc/jail.d/jailname.conf`. See [jail.conf\(5\)](#) for details.

The `vnetjail` configuration sets up a VNET network as follows:

```
#
```

```

# vnetjail.conf - handbook/jails - setting up a thin jail under ZFS
#
vnetjail {
  # Startup / Logging
  exec.start = "/bin/sh /etc/rc";
  exec.stop = "/bin/sh /etc/rc.shutdown";
  exec.consolelog = "/var/log/jail_console_${name}.log";

  # Permissions
  allow.raw_sockets;
  exec.clean;
  mount.devfs;
  devfs_ruleset = 5;

  # Hostname / Path
  host.hostname = "${name}";
  path = "/usr/local/jails/containers/${name}";

  # VNET / VIMAGE
  vnet;
  vnet.interface = "${epair}b";

  # Network
  $id = "90";
  $ip = "203.0.113.${id}/24";
  $gateway = "203.0.113.50";
  $bridge = "bridge0";
  $epair = "epair${id}";

  # ADD TO bridge INTERFACE
  exec.prestart = "/sbin/ifconfig ${bridge} create up";
  exec.prestart += "/sbin/ifconfig ${epair} create up";
  exec.prestart += "/sbin/ifconfig ${epair}a up descr jail:${name}";
  exec.prestart += "/sbin/ifconfig ${bridge} addm ${epair}a up";
  exec.prestart += "/sbin/ifconfig ${bridge} addm em0";
  exec.start += "/sbin/ifconfig ${epair}b ${ip} up";
  exec.start += "/sbin/route add default ${gateway}";
  exec.poststop = "/sbin/ifconfig ${bridge} deletem ${epair}a";
  exec.poststop += "/sbin/ifconfig ${bridge} deletem em0";
  exec.poststop += "/sbin/ifconfig ${epair}a destroy";
  exec.poststop += "/sbin/ifconfig ${bridge} destroy";
}

```

In this instance, the network stack is completely separate from the host network stack. However, achieving and managing connectivity happens *inside* the **vnetjail** jail.

Testing connectivity with the jail can be accomplished by

1. Ensuring **ipfw** is not loaded on **jail1**,
2. Entering the **vnetjail**, and

3. Starting up a listening service using `nc(1)`:

```
root@jail1:~ # kldunload ipfw
IP firewall unloaded
root@jail1:~ # service jail onestart vnetjail
root@jail1:~ #
root@jail1:~ # jexec -u root vnetjail
root@vnetjail:/ # cd
root@vnetjail:~ #
root@vnetjail:~ # nc -l -k 5656
```

Connecting from `external1` using `nc(1)`:

```
root@external1:~ # nc 203.0.113.90 5656
hello from external1
^C
#
```

With no `ipfw` firewall in place, the test is successful.

To apply `ipfw` rules for the `vnetjail` jail, start `ipfw` in the `jail1` VM.



In VNET jails, `ipfw` is *started* from outside the jail, but rules are added from *inside* the jail. `ipfw` is also stopped from outside the jail.

Then, from inside the `vnetjail` jail, start up a listener using `nc(1)`:

```
root@vnetjail:~ # nc -l -k 5656
root@vnetjail:~ #
```

Since the `vnetjail` jail has a separate IP address and network stack from the `jail1` VM, orient `ipfw` rules around the `vnetjail` IP address:

```
root@jail1:~ # kldload ipfw
ipfw2 (+ipv6) initialized, divert loadable, nat loadable, default to deny, logging
disabled
root@jail1:~ #
root@jail1:~ # jexec -u root vnetjail
root@vnetjail:/ # cd
root@vnetjail:~ #
root@vnetjail:~ # ipfw show
65535 0 0 deny ip from any to any
root@vnetjail:~ #
root@vnetjail:~ # ipfw add 100 check-state
00100 check-state :default
root@vnetjail:~ #
```

```
root@vnetjail:~ # ipfw add 1000 allow tcp from any to me dst-port 5656 setup keep-
state
01000 allow tcp from any to me 5656 setup keep-state :default
root@vnetjail:~ #
root@vnetjail:~ # ipfw show
00100 0 0 check-state :default
01000 0 0 allow tcp from any to me 5656 setup keep-state :default
65535 0 0 deny ip from any to any
```

The single rule above is enough to set up a TCP connection.

From **external1**:

```
root@external1:~ # nc 203.0.113.90 5656
Hello from external1 after ipfw rules have been set up.
^C
root@external1:~ #
```

After the above:

```
root@vnetjail:~ #
root@vnetjail:~ # ipfw show
00100 0 0 check-state :default
01000 18 1012 allow tcp from any to me 5656 setup keep-state :default
65535 0 0 deny ip from any to any
root@vnetjail:~ #
```

Setting up rules in the **vnetjail** is left as an exercise to the reader.

Appendix A: Appendix A: QEMU Setup



It is highly advised to read through the entire installation procedure, including all notes and tips, before proceeding.

This appendix contains helpful information for getting QEMU installed and running on FreeBSD, and instructions for common use. The installation presumes a graphical desktop running on top of X Windows or Wayland. Because of this, the user environment, either X Window or Wayland, must allow for use of the `DISPLAY` variable in the local environment.

The examples in this book utilize virtual machines (VMs) using an SDL-based `vt(4)` console and also include a using a FreeBSD serial console. Instructions for setting up the VMs are below. Instructions for setting up and managing the serial consoles are found in [Adding and Managing Serial Console Access to the VMs](#) below.

Additional resources for understanding and using QEMU include:

- The [QEMU Virtualization](#) chapter in the FreeBSD Handbook
- The [qemu\(1\)](#) manual page
- The [QEMU Home Page](#)
- [Yet Another QEMU installation guide](#)

Installation

QEMU is available as a package or a port. There are a large number of build options on the port, so in most cases it is best to install the package. `sudo(8)` will also need to be installed as well.

Altogether, there are nine virtual machines used in this book. The following procedure will make ready all nine VMs. However, all nine are not needed immediately. Most of the first half of this book can be done with just the **firewall**, **internal**, **external1**, and **external2** VMs.

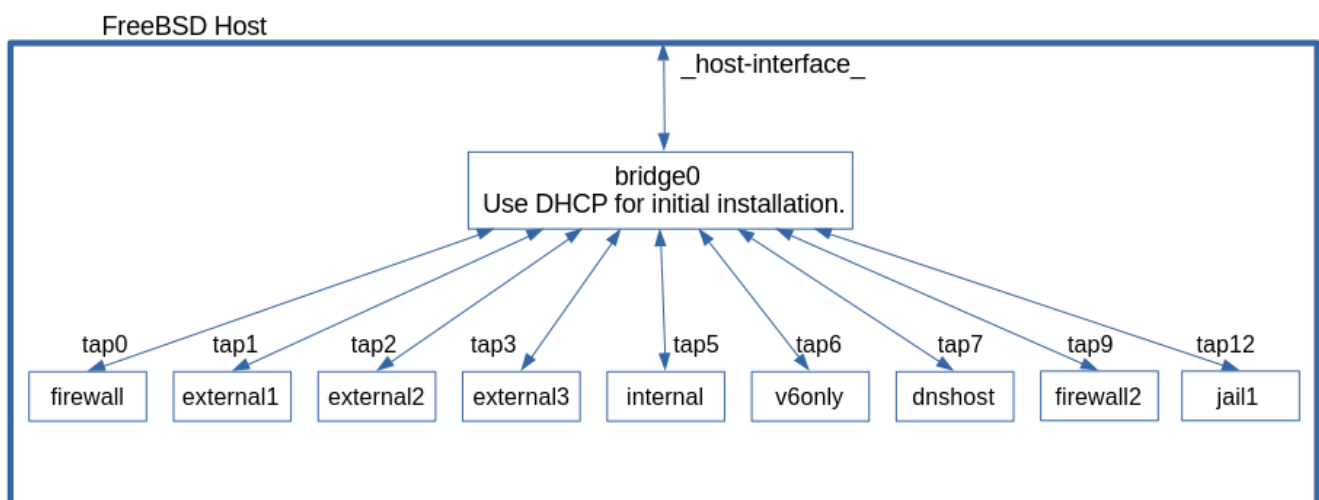


Figure 43. Setting Up the Initial Virtual Machines

The initial setup is that shown in [Figure 1](#).

A.1. QEMU and VM Installation Process

Follow the steps below to install and configure the QEMU virtual machines needed for this book. Various scripts are used to create virtual machines and set up bridge and tap devices. If desired, examine all scripts before use.

1. On the FreeBSD host, install the necessary packages - [qemu\(1\)](#), [sudo\(8\)](#) (or [doas\(1\)](#)). Sudo, (or doas) is necessary for running the virtual machines as these QEMU configurations open a separate console window through SDL. The examples in this book use sudo.

```
# pkg install qemu sudo
```

Configure sudo as desired.

2. Clone the scripts for this book from the **ipfw-primer** project on GitHub. Then fetch the current FreeBSD ISO.

Note: The initial path can be any directory.
All scripts use relative directory addressing.
Adjust the paths below as necessary.

```
% cd $HOME
% git clone https://github.com/jimmyb-gh/ipfw-primer.git
% cd ~/ipfw-primer/ipfw/ISO
% fetch https://download.freebsd.org/releases/amd64/amd64/ISO-IMAGES/<latest
version>/FreeBSD-<latest-version>-RELEASE-amd64-dvd1.iso
```

The example for FreeBSD 14.2 would be:

```
% fetch https://download.freebsd.org/releases/amd64/amd64/ISO-
IMAGES/14.2/FreeBSD-14.2-RELEASE-amd64-dvd1.iso
```

```
% # Link a shorter name to the ISO image.
```

```
% ln -s FreeBSD-<latest-version>-RELEASE-amd64-dvd1.iso fbsd.iso
```

3. Using **sudo**, create the bridge and tap devices for the virtual machines to use. See the description of the **mkbr.sh** script in [Using mkbr.sh for Bridge and Tap Setup](#).

```
% cd ../HOST_SCRIPTS
% sudo /bin/sh mkbr.sh reset bridge0 tap0 tap1 tap2 tap3 tap5 tap6 tap7 tap9
tap12 hostintf <--- replace hostintf with host network interface (em0, bge0,
etc.)
```


4. As a normal user, create all VM image files. Each VM is 4GB except for the **jail1** VM which is 12GB.

```
% /bin/sh _CreateAllVMs.sh
```

The next command starts the installation for the **firewall** VM.
Once finished, return to this point and repeat for the following VMs:
external1, external2, external3, firewall2, internal, dnshost, v6only.

```
% sudo /bin/sh firewall.sh
```

Ignore the "NOTE!!! telnet server running..." message for now.

Instructions for setting up a serial console are found later in this setup guide.

The FreeBSD installer should boot. Perform a standard installation of FreeBSD.

During the installation note the following:

- On all VMs *except* the **jail1** VM, select to use UFS as the filesystem. ZFS does not perform well with small memory sizes. The **jail1** VM has more memory and ZFS will be used to create jails on the **jail1** VM. It is the only VM that requires ZFS.
- For these installations, use DHCP for networking. If desired, configure IPv6 if supported by the local LAN.
- When adding the default user, ensure they are a member of the *wheel* group.

Once the installation completes, the virtual machine reboots into the newly installed FreeBSD image.

5. Login as root, update the system, and reboot.

```
# freebsd-update fetch install  
# reboot
```

6. On all virtual machines, install the packages listed below. The **nmap** package brings in the version of **ncat(1)** used by scripts on the **firewall** and **external** VMs. **nginx**, **lynx**, **cmdwatch**, **hping3**, **tsctp**, and **iperf3** will be used in later chapters.

```
# pkg install nmap nginx lynx cmdwatch hping3 tsctp iperf3
```

7. Finally, download `~/ipfw-primer/ipfw/VM_SCRIPTS/IPFW_root_bin.tgz` file to *all* VMs. This tar file has a number of scripts needed for the virtual machines.

Move the tarzip file into /root and extract the contents:

On each VM, login as root, copy and untar the following file:

```
# scp user@hostip:~/ipfw-primer/ipfw/VM_SCRIPTS/IPFW_root_bin.tgz .
#
# mv IPFW_root_bin.tgz /root
#
# cd /root
#
# tar xvzf IPFW_root_bin.tgz
... files are extracted into /root/bin
#
# chmod +x /root/bin/*.sh
```

8. Repeat the installation procedure for each virtual machine.

Some additional configurations are required for examples later in the book:

On the **firewall** and **firewall2** VMs:

- Add **net.inet.ip.fowarding=1** and **net.inet6.ip6.fowarding=1** to **/etc/sysctl.conf**.

On the **external1** and **internal** VMs install these extra packages:

- **pkg install git**
- **pkg install cmake**

On each VM, navigate to /usr/local/www/nginx. Download the modified **index.html** file from the **host** system to replace the original:

- **scp user@host:~/ipfw-primer/ipfw/VM_SCRIPTS/VM_name/index.html .**

On the **firewall** VM, download the **bsdclat464.sh** script

- **cd /root/bin**
- **scp user@host:~/ipfw-primer/ipfw/VM_SCRIPTS/firewall/bsdclat464.sh .**

On the ***firewall2** VM, download the **bsdplat464.sh** script

- **cd /root/bin**
- **scp user@host:~/ipfw-primer/ipfw/VM_SCRIPTS/firewall/bsdplat464.sh .**

(End installation procedure.)

For this Quick Start, it is Ok to use **DHCP** for both VMs. In later examples there will be multiple external VMs using the **203.0.113.0/24** network and other private networks, all set up the same way and attached via **tap(4)** interfaces to one or more **if_bridge(4)** interfaces on the FreeBSD host.

To ensure the first two VMs are set up correctly, ping the **firewall** VM from the **external1** VM and vice-versa. Communications should be successful. If not, check the above installation details and troubleshoot any network issues. It should be possible ping in both directions, and even **ssh(1)** from one VM to the other.



For additional helpful information on getting QEMU set up correctly, check the [QEMU virtualization section in the FreeBSD Handbook](#).



If the mouse is clicked in the QEMU console window, QEMU will “grab” the mouse. If this happens, type, **Ctrl + Alt + G** to release the mouse.



If suddenly, the QEMU console window is full screen, you may have accidentally typed **Ctrl + Alt + F**. If this happens, retype **Ctrl + Alt + F** to restore the desktop screen.

A.1.1. Disabling Syslog Messages to the Console in the Virtual Machines

It may be advantageous (even necessary) to stop syslog messages from being sent to the console (either the QEMU console, or the serial port).

To configure syslog to stop logging to the console, configure a file to receive console messages:

```
# touch /var/log/console.log
#
# chmod 0600 /var/log/console.log
```

Then, as **root**, modify the line in `/etc/syslog.conf` to read (instead of `/dev/console`):

```
*.err;kern.warning;auth.notice;mail.crit    /var/log/console.log
```

And, if necessary:

```
# service syslogd restart
```

All messages previously bound for the console, will be directed to `/var/log/console.log` instead.

Before continuing, there is one more piece to add to each VM - a serial console. A serial console permits examination of the state of each VM, independent of the main console.

A.1.2. Adding and Managing Serial Console Access to the VMs

Adding a serial console the FreeBSD VM

To add a serial console to each FreeBSD VM, start up the VM and edit the file `/boot/loader.conf` and add the line `console="comconsole"` to allow use of the serial console. Reboot the VM to begin using the serial console. Note that FreeBSD diverts boot I/O to the serial console, so until the FreeBSD operating system is completely ready, output to the QEMU window will be limited.

The startup scripts for each VM are already configured to use a serial console. A single configuration line was added to the QEMU configuration to provide a serial console. The serial console is actually accessed over a [telnet\(1\)](#) session. The QEMU manual page, [qemu\(1\)](#), describes how the `-serial` keyword works in detail.

QEMU redirects the serial port I/O to a TCP port on the host system at VM startup, and allows a [telnet\(1\)](#) connection on the configured port on the host. Once the FreeBSD system starts booting and recognizes the console directive in `/boot/loader.conf` it redirects I/O to the serial console. QEMU detects this and manages the necessary character I/O on that serial port to the TCP port on the host.

It is important to note that the this serial redirect over TCP takes place outside the virtual machine. There is no interaction with any network on the virtual machine and thus it is not subject to any firewall rules. Think of it just like a "dumb terminal" sitting on an RS-232 serial port on a real machine.

Management of serial console windows on the FreeBSD host

Each QEMU VM generates a console window, and each serial device also needs its own window, potentially doubling the number of windows used.

Possible solutions are:

- Separate windows for each QEMU VM (doubles the number of windows)
- Use tabbed windows (available on XFCE and some other desktops)
- Use a terminal multiplexer such as [tmux\(1\)](#) or [screen\(1\)](#)

The selected solution uses the multiplexer approach with the [tmux\(1\)](#) program for window management. [Appendix D](#) provides details on using both [tmux\(1\)](#) and [screen\(1\)](#). The following figures and descriptions use [tmux\(1\)](#).

Install `tmux` on the FreeBSD `host` with:

```
# pkg install tmux
```

and if necessary, copy the file `swim.sh` (or `scim.sh` for using [screen\(1\)](#)) from [Appendix B](#) into the `HOST_SCRIPTS` directory. If using `scim.sh`, follow the instructions in the script to set up a `.screenrc`

file.

The figure below shows the use of the **swim.sh** tmux session manager. Run **sh swim.sh** in the **HOST_SCRIPTS** directory to start up the session manager.

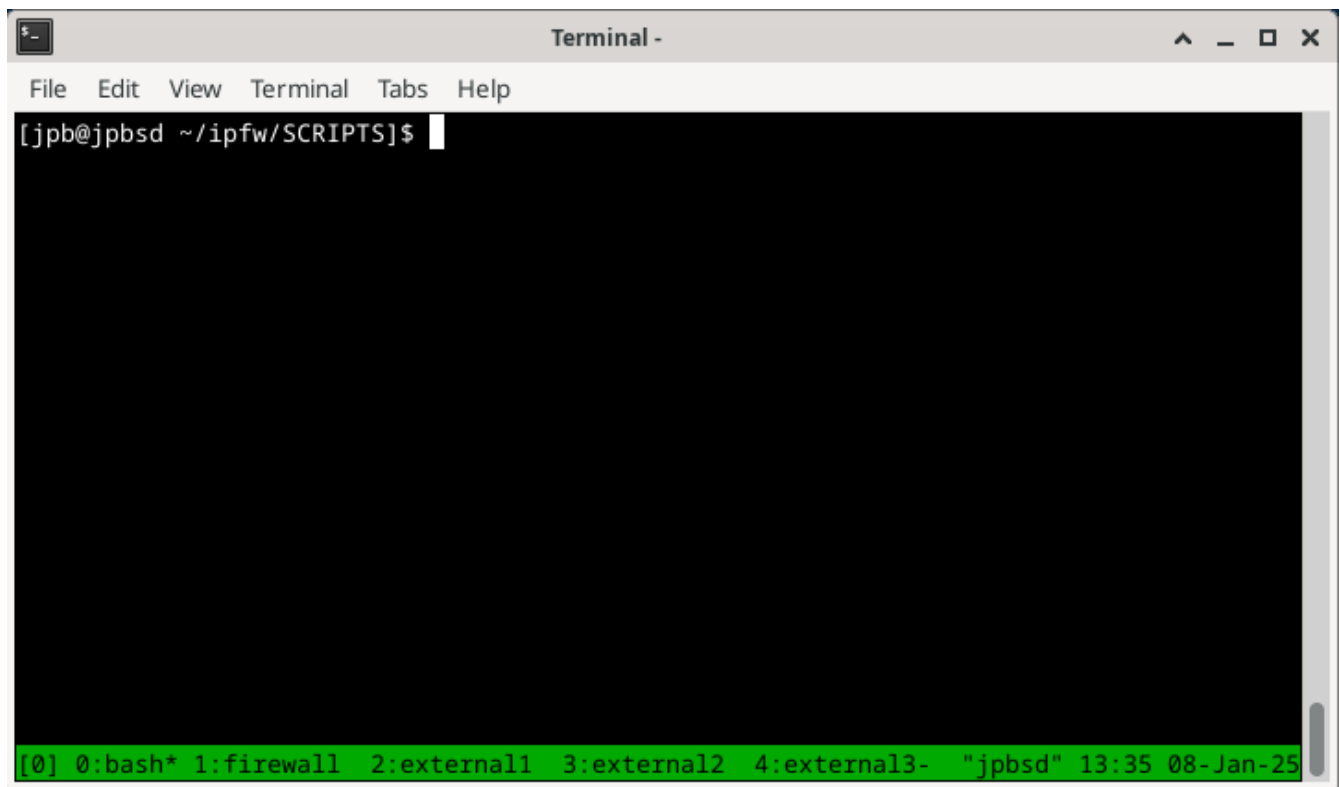


Figure 44. Starting Up tmux(1) Session Manager

The figure shows five named windows in one session (session [0]) with the **tmux** status line in green at the bottom:

- **0:bash** - a terminal window of the user running **swim.sh**
- **1:firewall** - a terminal window to access the **firewall** VM
- **2:external1** - a terminal window to access the **external1** VM
- **3:external2** - a terminal window to access the **external2** VM
- **4:external3** - a terminal window to access the **external3** VM

The current window is marked with the '*' character on the status bar.

Simplified tmux(1) Usage

tmux uses **Ctrl+b** as its control key. To move from window to window use **Ctrl+b n** to move to the next window or **Ctrl+b p** to move to the previous window. Use **Ctrl+b ?** for a list of all key bindings.

Type **tmux kill-server** in any host session shell window (not a VM window) to completely leave **tmux**.

Consult the **tmux** manual page [tmux\(1\)](#) for more usage details.

Accessing the QEMU Serial Consoles

To access the VM serial consoles, move to the indicated window and telnet to the port on the local host for that VM:

```
Move to the external1 window in tmux, then
```

```
% telnet localhost 4410
Trying ::1...
Connected to localhost.
Escape character is '^]'.
```

```
FreeBSD/amd64 (external1) (ttyu0)
```

```
login:
```

To exit out of the **telnet** session, press `Ctrl+]` then press `q` like this:



```
login: (type Ctrl+])
telnet> q
Connection closed.
%
```

There should now be two QEMU VMs (**firewall** and **external1**) started, with serial console sessions available through the **tmux** sessions as shown below.

Configure the FreeBSD **host**, **firewall** VM, and **external1** VM with DHCP addressing as shown in the figure at the beginning of this Quick Start session. There should be full connectivity between the FreeBSD **host**, the **firewall** VM and the **external1** VM.

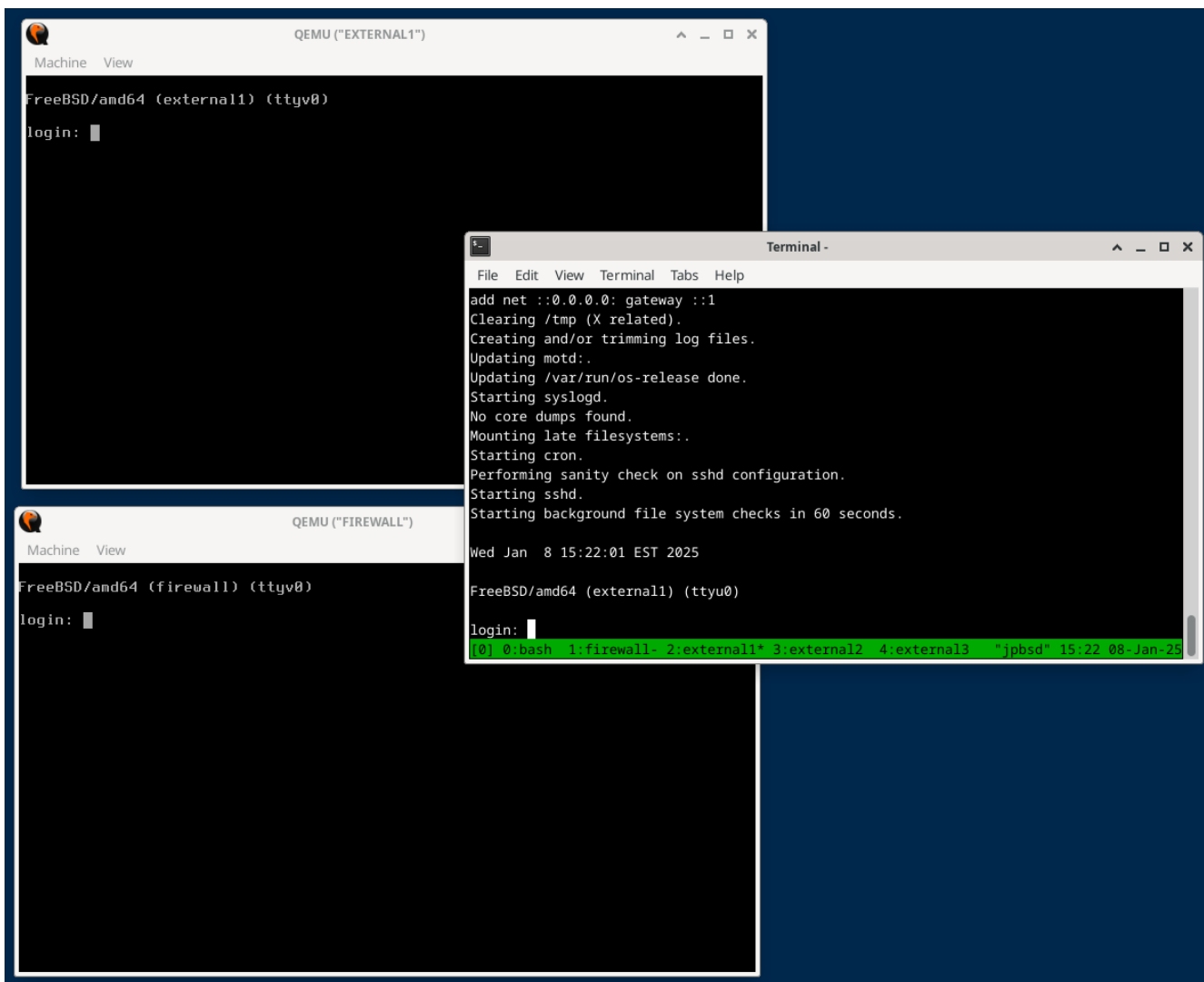


Figure 45. External1 and Firewall VMs Startup with Serial Console

Using **ipfw** to control traffic between these two QEMU VMs is discussed in the next chapter.

A.2. Using **mkbr.sh** for Bridge and Tap Setup

Included in [Appendix B](#), the **mkbr.sh** script is used to set up **if_bridge(4)** and **tap(4)** devices on the FreeBSD host. These interfaces are required to allow the virtual machines to communicate with each other and, when suitably configured, to communicate with the outside world.

Many examples in this book include a statement such as:

```
# /bin/sh mkbr.sh reset bridge0 tap0 tap1 em0
```

or something similar.

The above invocation resets the kernel modules necessary for bridge and tap operation, then it creates one bridge, **bridge0**, and connects **tap0**, **tap1**, and **em0**. Here, "em0" refers to a host ethernet interface, but it can be any ethernet interface on the local machine.

The script can be used to create any number of bridges and taps for complex network designs. For example, the following invocation creates three bridges - **bridge0** with **tap0** and **tap1** connected,

bridge1 with tap2, tap3, and tap4 connected, and bridge2 with tap5 and host interface igb0 connected.

```
% sudo /bin/sh mkbr.sh reset bridge0 tap0 tap1 bridge1 tap2 tap3 tap4 bridge2 tap5
igb0
```

To add other taps to existing bridges, do not specify the "reset" parameter:

```
# /bin/sh mkbr.sh bridge0 tap10 tap11 bridge1 tap12 tap13 ... etc.
```

To delete all bridge and tap devices:

```
# /bin/sh mkbr.sh reset
```

Usually, the examples include an architecture diagram, like that shown earlier, and an invocation of **mkbr.sh** that will create the architecture shown.

Below is a handy chart to show the relationship of virtual machines and **tap** devices. Note that some virtual machines have more than one interface. The chart also shows how all virtual machines could be attached to the same bridge if needed for administrative purposes.

TAPLIST.TXT

This file contains just the taps that are needed to connect all VMs to one bridge for admin purposes:

```
+-----+
|                                     |
|               bridge0              |
|                                     |
+-----+
/ / / / | | \ \ \ \
tap0 tap1 tap2 tap3 tap5 tap6 tap7 tap9 tap12 host_interface
| | | | | | | | | |
| | | | | | | | +- jail1:em0
| | | | | | | | +- firewall2:em0
| | | | | | | | +- dnshost:em0
| | | | | | +- v6only:em0
| | | | +- internal:em0
| | | +- external3:em0
| | +- external2:em0
| +- external1:em0
+- firewall:em0
```

The remaining taps are located on the VM listed:


```
tap4 - firewall:em1
tap8 - dnshost:em1
tap10 - firewall2:em1
tap11 - dnshost:em2
```

Appendix B: Appendix B: Scripts and Code for QEMU Lab

The listing below shows how the scripts are organized on the [GitHub ipfw-primer/SCRIPTS](#) site.

```
.
|-- VM_SCRIPTS
| |-- IPFW_root_bin.tgz           : common scripts for all VMs (see bin
list)
| |-- Manifest_IPFW_root_bin.txt  : Manifest document for IPFW_root_bin.tgz
| |-- Manifest_index.txt         : Manifest document for index.html files
|-- bin
| |-- tcon.sh                    : TCP connection script
| |-- tconr.sh                   : TCP connect with random port script
| |-- tcont.sh                   : TCP continuous connection script
| |-- tserv.sh                   : TCP server script for one port
| |-- tserv3.sh                  : TCP server script for three ports
| |-- ucon.sh                    : UDP connection script
| |-- uconr.sh                   : UDP connect with random port script
| |-- ucont.sh                   : UDP continuous connection script
| |-- userv.sh                   : UDP server script for one port
| |-- userv3.sh                  : UDP server script for three ports
| |-- `-- userv5.sh              : UDP server script for five ports
| |-- dnshost                    :
| | |-- Manifest_namedb.txt      : Manifest for dnshost
/usr/local/etc/namedb
| | |-- dnshost_usrlocaletc_namedb.tgz : Files for the above
| | |-- `-- index.html          : Nginx index.html file for dnshost VM
| |-- external1                  :
| | |-- `-- index.html          : Nginx index.html file for external1 VM
| |-- external2                  :
| | |-- `-- index.html          : Nginx index.html file for external2 VM
| |-- external3                  :
| | |-- `-- index.html          : Nginx index.html file for external3 VM
| |-- firewall                   :
| | |-- |-- bsdclat464.sh        : Script for Section 6.2 XLAT464 CLAT
| | | |-- `-- index.html        : Nginx index.html file for firewall VM
| |-- firewall2                  :
| | |-- |-- bsdplat464.sh        : Script for Section 6.2 XLAT464 CLAT
| | | |-- `-- index.html        : Nginx index.html file for firewall2 VM
| |-- internal                   :
| | |-- `-- index.html          : Nginx index.html file for internal VM
| |-- v6only                     :
| | |-- `-- index.html          : Nginx index.html file for v6only VM
| |-- `-- jail1                  :
| | |-- `-- index.html          : Nginx index.html file for jail1 VM
|-- _CreateAllVMs.sh            : Script to create all VMs used in book
|-- dnshost.sh                  : QEMU startup script for dnshost VM
|-- external1.sh                : QEMU startup script for external1 VM
```

```

|-- external2.sh           : QEMU startup script for external2 VM
|-- external3.sh           : QEMU startup script for external3 VM
|-- firewall.sh            : QEMU startup script for firewall VM
|-- firewall2.sh           : QEMU startup script for firewall2 VM
|-- internal.sh            : QEMU startup script for internal VM
|-- jail1.sh               : QEMU startup script for jail1 VM
|-- mkbr.sh                : Script to make host bridge and tap
devices
|-- runvm.sh               : XFCE4 script to start VMs
|-- swim.sh                : Script to manage serial terminals on
host
|-- scim.sh                : Script to manage serial terminals on
host
|-- v6only.sh              : QEMU startup script for v6only VM
|-- vm_envs.sh             : IPFW lab environment variables
`-- CODE
    `-- divert.c           : C code for working with divert keyword

```

All scripts are shown below in lexicographic order:

SCRIPT: VM_SCRIPTS/firewall/bsdclat464.sh

```

#!/bin/sh
# IPFW Primer
# License: 3-clause BSD
# Author: Jim Brown, jpb@jimby.name
# Code: https://github.com/jimmyb-gh/ipfw-primer
#
# FreeBSD 464XLAT CLAT script for firewall VM.
#
# bsdclat464.sh: FreeBSD IPFW script for 464XLAT CLAT. See Section 6.2
# Usage: # /bin/sh bsdclat464.sh (run script as root)

set -x

kldunload ipfw_nat64
kldunload ipfw
sleep 1
kldload ipfw
kldload ipfw_nat64

# Create the nat64clat instance
ipfw nat64clat CLAT create clat_prefix 2001:db8:aaaa::/96 plat_prefix
2001:db8:bbbb::/96 allow_private log

# Allow neighbor discovery
ipfw add 100 allow log icmp6 from any to any icmp6types 135,136

```

```

# pass any ip through the nat64clat instance
ipfw add 150 nat64clat CLAT log ip from any to any

# pass any ip through the nat64plat instance
ipfw add 200 nat64clat CLAT log ip from any to 2001:db8:bbbb::/96

# allow ipv6 from any to any
ipfw add 300 allow log ip6 from any to any

# allow ipv4 from any to any
ipfw add 400 allow log ip from any to any

# 0=log with ipfwlog0, 1=log with syslog
sysctl net.inet.ip.fw.verbose=0

sysctl net.inet.ip.fw.nat64_debug=1

# direct output: 1 enable, 0 disable (packet goes back into ruleset)
sysctl net.inet.ip.fw.nat64_direct_output=1

indexterm:[bsdclat464.sh]

=====

SCRIPT: VM_SCRIPTS/firewall12/bsdplat464.sh

#!/bin/sh
# IPFW Primer
# License: 3-clause BSD
# Author: Jim Brown, jpb@jimby.name
# Code: https://github.com/jimmyb-gh/ipfw-primer
#
# FreeBSD 464XLAT PLAT script for firewall12 VM.
#
# bsdplat464.sh: FreeBSD IPFW script for 464XLAT CLAT. See Section 6.2
# Usage: # /bin/sh bsdplat464.sh (run script as root)

set -x

kldunload ipfw_nat64
kldunload ipfw

sleep 1

kldload ipfw
kldload ipfw_nat64

# create the nat64 stateful instance
ipfw nat64lsn NAT64 create log prefix4 203.0.112.0/24 prefix6 2001:db8:bbbb::/96
allow_private

```

```

# Allow neighbor discovery
ipfw add allow log icmp6 from any to any icmp6types 135,136

# Allow the nat64 outbound
ipfw add nat64lsn NAT64 log ip from 2001:db8:12::/64 to 2001:db8:bbbb::/96 in

ipfw add nat64lsn NAT64 log ip from any to 2001:db8:bbbb::/96 in

# Allow the nat64 inbound
ipfw add nat64lsn NAT64 log ip from any to 203.0.112.0/24 in

# Allow ipv4 from any to any
ipfw add allow log ip from any to any

# Allow ipv6 from any to any
ipfw add allow log ip6 from any to any

# Logging: 0 interfaces, 1 syslog
sysctl net.inet.ip.fw.verbose=0

# Debug nat64
sysctl net.inet.ip.fw.nat64_debug=1

# Direct output: 1 enable, 0 disable (packet goes back into ruleset)
sysctl net.inet.ip.fw.nat64_direct_output=1

indexterm:[bsdplat464.sh]

```

=====

SCRIPT: _CreateAllVMs.sh

```

#!/bin/sh
# IPFW Primer
# License: 3-clause BSD
# Author: Jim Brown, jpb@jimby.name
# Code: https://github.com/jimmyb-gh/ipfw-primer
#
# _CreateAllVM.sh : Create VMs for the IPFW Primer lab.
# Files are created in ../VM/
#

echo "Running _CreateAllVMs.sh"

echo
echo "This script will create 8 virtual machines in ../VM/"
echo
read -p "DO YOU REALLY WANT TO CREATE NEW QEMU IMAGES OVERWRITING ANY EXISTING IMAGES?
Answer YES to continue. " junk

```

```

echo [${junk}]

if [ "X${junk}" != "XYES" ]
then
    echo "Response was [${junk}]"
    echo "bailing out..."
    exit 1
fi

echo "Response was [${junk}]"
echo "Ok, continuing..."

#exit

for i in dnshost external1 external2 external3 firewall firewall2 internal v6only
do
    echo "Creating ${i} VM"
    echo qemu-img create -f qcow2 -o preallocation=full ../VM/${i}.qcow2 4G
    qemu-img create -f qcow2 -o preallocation=full ../VM/${i}.qcow2 4G
done

echo
echo "Done."

```

=====

SCRIPT: tcon.sh

```

#!/bin/sh
#
# location: external Vms
#
# sh tcon.sh PORTNUM - start up 1 connection over TCP
#

usage() {
    echo "sh tcon.sh PORTNUM"
    exit 1
}

#echo $#

if [ $# -ne 1 ]
then
    usage
else

```

```

export PORT1=$1
fi

# echo "PORT1 = [$PORT1]"

export CONN="203.0.113.50"
export COUNT=1

export MYIP=ifconfig em0 | grep inet | grep -v inet6 | awk '{print $2}'
export MYNAME="external1"

echo "TCP connection from [$MYIP],[$PORT1],[$COUNT]"
echo "TCP connection from [$MYIP],[$PORT1],[$COUNT]" | ncat $CONN $PORT1

export PREVIOUS_PORT=$PORT1

while :
do

    COUNT=expr $COUNT + 1

    read -p "ncat [$COUNT] ready. Enter a valid PORTNUM: " PORT1

    if [ "$PORT1" = "X" ]
    then
        PORT1=$PREVIOUS_PORT
    fi

    echo "TCP connection from [$MYIP],[$PORT1],[$COUNT]"
    echo "TCP connection from [$MYIP],[$PORT1],[$COUNT]" | ncat $CONN $PORT1

    if [ $? -ne 0 ]
    then
        echo "TCP connection [$MYIP],[$PORT1],[$COUNT] FAILED"
    fi

    PREVIOUS_PORT=$PORT1

done

=====

SCRIPT: tconr.sh

#!/bin/sh
#
# location: external Vms
#
# sh tconr.sh PORTNUM SLEEPVAL (randomized port numbers) - start up 1 connection over

```

```

TCP
#

usage() {
    echo "sh tconr.sh PORT1NUM SLEEPVAL (randomized port numbers)"
    exit 1
}

# echo $#

if [ $# -ne 2 ]
then
    usage
fi

PORT1=$1
SLEEPVAL=$2

echo "PORT1      = [$PORT1]"
echo "SLEEPVAL = [$SLEEPVAL]"

export CONN="203.0.113.50"
export COUNT=1
export MYIP=ifconfig em0 | grep inet | grep -v inet6 | awk '{print $2}'
export MYNAME="external1"

echo "TCP connection from [$MYIP],[$PORT1],[$COUNT]"
echo "TCP connection from [$MYIP],[$PORT1],[$COUNT]" | ncat $CONN $PORT1

while :
do

COUNT=expr $COUNT + 1

# use jot(1) to get a random port between 5656 and 5659.
# Connection to 5659 has no listener on firewall and will thus fail.

PORT1=jot -r 1 5656 5659 $RANDOM

echo "TCP connection from [$MYIP],[$PORT1],[$COUNT]"
echo "TCP connection from [$MYIP],[$PORT1],[$COUNT]" | ncat $CONN $PORT1

if [ $? -ne 0 ]
then
    echo "TCP connection [$MYIP],[$PORT1],[$COUNT] FAILED"
fi

sleep $SLEEPVAL

done

```



```
=====
SCRIPT: tcont.sh
```

```
#!/bin/sh
#
# location: external Vms
#
# sh tcont.sh PORT1NUM SLEEPVAL - keep hammering same TCP port every SLEEPVAL
#
```

```
usage() {
    echo "sh tcont.sh PORT1NUM SLEEPVAL"
    exit 1
}
```

```
#echo $#
```

```
if [ $# -ne 2 ]
then
    usage
fi
```

```
export PORT1=$1
export SLEEPVAL=$2
```

```
echo "PORT1 = [$PORT1]"
echo "SLEEPVAL = [$SLEEPVAL]"
```

```
export CONN="203.0.113.50"
export COUNT=1
export MYIP=ifconfig em0 | grep inet | grep -v inet6 | awk '{print $2}'
```

```
echo "TCP connection from [$MYIP],[$PORT1],[$COUNT]"
echo "TCP connection from [$MYIP],[$PORT1],[$COUNT]" | ncat $CONN $PORT1
```

```
while :
do
```

```
    COUNT=expr $COUNT + 1
```

```
    echo "TCP connection from [$MYIP],[$PORT1],[$COUNT]"
    echo "TCP connection from [$MYIP],[$PORT1],[$COUNT]" | ncat $CONN $PORT1
```

```
    if [ $? -ne 0 ]
    then
```

```
    echo "TCP connection [$MYIP],[\$PORT1],[\$COUNT] FAILED"
fi

sleep \$SLEEPVAL

done
```

=====

SCRIPT: tserv.sh

```
#!/bin/sh
#
# location: firewall VMs
#
# tserv.sh - start up 1 listener over TCP

zapall() {
kill -TERM $PID1
}

trap zapall SIGINT

export MYIP=ifconfig em0 | grep inet | grep -v inet6 | awk '{print $2}'

export PORT1=5656

echo "Starting TCP listener on [\$PORT1]"

ncat -l -4 -k $MYIP $PORT1 &
PID1=$!

wait

exit
```

=====

SCRIPT: tserv3.sh

```
#!/bin/sh
#
# location: firewall VMs
#
# tserv3.sh - start up 3 listeners over TCP
```

```

zapall() {
kill -TERM $PID1 $PID2 $PID3
}

trap zapall SIGINT

export MYIP=ifconfig em0 | grep inet | grep -v inet6 | awk '{print $2}'

export PORT1=5656
export PORT2=5657
export PORT3=5658

echo "Starting TCP listeners on [$PORT1],[$PORT2],[$PORT3]"

ncat -l -4 -k $MYIP $PORT1 &
PID1=$!

ncat -l -4 -k $MYIP $PORT2 &
PID2=$!

ncat -l -4 -k $MYIP $PORT3 &
PID3=$!

wait

exit

```

=====

SCRIPT: ucon.sh

```

#!/bin/sh
#
# location: external Vms
#
# sh ucon.sh PORTNUM - start up 1 transfer over UDP
#

usage() {
    echo "sh ucon.sh PORTNUM"
    exit 1
}

#echo $#

if [ $# -ne 1 ]
then
    usage

```

```

else
  export PORT1=$1
fi

# echo "PORT1 = [$PORT1]"

export CONN="203.0.113.50"
# export CONN="10.10.10.50"
export COUNT=1

export MYIP=ifconfig em0 | grep inet | grep -v inet6 | awk '{print $2}'
export MYNAME="external1"

echo "UDP packet from [$MYIP],[$PORT1],[$COUNT]"
echo "UDP packet from [$MYIP],[$PORT1],[$COUNT]" | ncat -u $CONN $PORT1

export PREVIOUS_PORT=$PORT1

while :
do

  COUNT=expr $COUNT + 1

  read -p "ncat [$COUNT] ready. Enter a valid PORTNUM: " PORT1

  if [ "$PORT1" = "X" ]
  then
    PORT1=$PREVIOUS_PORT
  fi

  echo "UDP packet from [$MYIP],[$PORT1],[$COUNT]"
  echo "UDP packet from [$MYIP],[$PORT1],[$COUNT]" | ncat -u $CONN $PORT1

  if [ $? -ne 0 ]
  then
    echo "UDP packet [$MYIP],[$PORT1],[$COUNT] FAILED"
  fi

  PREVIOUS_PORT=$PORT1

done

=====

SCRIPT: uconr.sh

#!/bin/sh
#
# location: external Vms

```

```

#

usage() {
    echo "sh uconr.sh PORT1NUM SLEEPVAL (randomized port numbers)"
    exit 1
}

# echo $$

if [ $# -ne 2 ]
then
    usage
fi

PORT1=$1
SLEEPVAL=$2

echo "PORT1      = [$PORT1]"
echo "SLEEPVAL = [$SLEEPVAL]"

export CONN="203.0.113.50"
export COUNT=1
export MYIP=ifconfig em0 | grep inet | grep -v inet6 | awk '{print $2}'
export MYNAME="external1"

echo "UDP packet from [$MYIP],[$PORT1],[$COUNT]"
echo "UDP packet from [$MYIP],[$PORT1],[$COUNT]" | ncat -u $CONN $PORT1

while :
do

COUNT=expr $COUNT + 1

# use jot(1) to get a random port between 5656 and 5659.
# Packet on 5659 has no listener on firewall and will thus fail.

PORT1=jot -r 1 5656 5659 $RANDOM

echo "UDP packet from [$MYIP],[$PORT1],[$COUNT]"
echo "UDP packet from [$MYIP],[$PORT1],[$COUNT]" | ncat -u $CONN $PORT1

if [ $? -ne 0 ]
then
    echo "UDP packet [$MYIP],[$PORT1],[$COUNT] FAILED"
fi

```

```

sleep $SLEEPVAL

done

=====

SCRIPT: ucont.sh

#!/bin/sh
#
# location: external Vms
#
# sh ucont.sh PORT1NUM SLEEPVAL - keep hammering same UDP port every SLEEPVAL
#

usage() {
    echo "sh ucont.sh PORT1NUM SLEEPVAL"
    exit 1
}

#echo $$

if [ $# -ne 2 ]
then
    usage
fi

export PORT1=$1
export SLEEPVAL=$2

echo "PORT1 = [$PORT1]"
echo "SLEEPVAL = [$SLEEPVAL]"

# export CONN="10.10.10.50"
export CONN="203.0.113.50"
export COUNT=1
export MYIP=ifconfig em0 | grep inet | grep -v inet6 | awk '{print $2}'

echo "UDP packet from [$MYIP],[PORT1],[COUNT]"
echo "UDP packet from [$MYIP],[PORT1],[COUNT]" | ncat -u $CONN $PORT1

while :
do

    COUNT=expr $COUNT + 1

# echo "UDP packet from [$MYIP],[PORT1],[COUNT]"
# echo "UDP packet from [$MYIP],[PORT1],[COUNT]" | ncat -u $CONN $PORT1

```

```

echo "UDP packet from [$MYIP],[$PORT1],[$COUNT]"
echo "UDP packet from [$MYIP],[$PORT1],[$COUNT]"| ncat -u $CONN $PORT1

if [ $? -ne 0 ]
then
    echo "UDP packet [$MYIP],[$PORT1],[$COUNT] FAILED"
fi

sleep $SLEEPVAL

done

```

=====

SCRIPT: userv.sh

```

#!/bin/sh
#
# location: firewall VMs
#
# userv.sh PORTNUM - start up 1 listener over UDP
#

usage() {
    echo "sh userv.sh PORTNUM"
    exit 1
}

#echo $#

if [ $# -ne 1 ]
then
    usage
else
    PORT1=$1
fi

echo "PORT1 = [$PORT1]"

zapall() {
kill -TERM $PID1
}

trap zapall SIGINT

export MYIP=ifconfig em0 | grep inet | grep -v inet6 | awk '{print $2}'

```

```
echo "Starting UDP listener on [${MYIP},${PORT1}]"
```

```
# echo nc -l -k -u ${MYIP} ${PORT1}
nc -l -k -u ${MYIP} ${PORT1} &
PID1=$!
```

```
wait
```

```
exit
```

```
=====
```

SCRIPT: user3.sh

```
#!/bin/sh
```

```
#
```

```
# location: firewall VMs
```

```
#
```

```
# user3.sh - start up 3 listeners over udp
```

```
zapall() {
```

```
kill -TERM $PID1 $PID2 $PID3
```

```
}
```

```
trap zapall SIGINT
```

```
export MYIP=ifconfig em0 | grep inet | grep -v inet6 | awk '{print $2}'
```

```
export PORT1=5656
```

```
export PORT2=5657
```

```
export PORT3=5658
```

```
echo "Starting UDP listeners on [${PORT1},${PORT2},${PORT3}]"
```

```
nc -l -k -u ${MYIP} ${PORT1} &
PID1=$!
```

```
nc -l -k -u ${MYIP} ${PORT2} &
PID2=$!
```

```
nc -l -k -u ${MYIP} ${PORT3} &
PID3=$!
```

```
wait
```

```
exit
```



```
=====
```

SCRIPT: userv5.sh

```
#!/bin/sh
#
# location: firewall VMs
#
# userv5.sh - start up 5 listeners over udp

zapall() {
kill -TERM $PID1 $PID2 $PID3 $PID4 $PID5
}

trap zapall SIGINT

export MYIP=$(ifconfig em0 | grep inet | grep -v inet6 | awk '{print $2}')
export PORT1=5656
export PORT2=5657
export PORT3=5658
export PORT4=5659
export PORT5=5660

echo "Starting UDP listeners on [$PORT1],[$PORT2],[$PORT3],[$PORT4],[$PORT5]"

nc -l -k -u $MYIP $PORT1 &
PID1=$!

nc -l -k -u $MYIP $PORT2 &
PID2=$!

nc -l -k -u $MYIP $PORT3 &cd
PID3=$!

nc -l -k -u $MYIP $PORT4 &
PID4=$!

nc -l -k -u $MYIP $PORT5 &
PID5=$!

wait

exit
```

```
=====
```

SCRIPT: dnshost.sh

```
#!/bin/sh
# IPFW Primer
# License: 3-clause BSD
# Author: Jim Brown, jpb@jimby.name
# Code: https://github.com/jimmyb-gh/ipfw-primer
#
# FreeBSD QEMU VM startup script for dnshost VM.
#
# dnshost.sh: FreeBSD QEMU VM startup script for dnshost VM.
# Usage: sudo /bin/sh dnshost.sh
# Note: Set up for serial console. Start another session and telnet to the port shown.
#
#set -x

# pick up environment for this run
. ./vm_envs.sh

echo [ISO=${_DNSHOST_ISO}]
echo [mem=${_DNSHOST_mem}]
echo [hdsiz=${_DNSHOST_hdsiz}]
echo [img=${_DNSHOST_img}]
echo [mac1=${_DNSHOST_mac1}]
echo [mac2=${_DNSHOST_mac2}]
echo [name=${_DNSHOST_name}]
echo [tap7=${_DNSHOST_tap7}]
echo [tap8=${_DNSHOST_tap8}]
echo [tap11=${_DNSHOST_tap11}]
echo [telnetport=${_DNSHOST_telnetport}]

#exit

# Note - the dnshost has two interfaces - em0 and em1.
#       em0 is considered the ipv4 interface and
#       em1 is considered the ipv6 interface.

echo
echo "NOTE!!! telnet server running! To start QEMU telnet to localhost
${_DNSHOST_telnetport}"
echo

/usr/local/bin/qemu-system-x86_64 -monitor none \
  -serial telnet:localhost:${_DNSHOST_telnetport},server=on,wait=off \
  -cpu qemu64 \
  -vga cirrus \
  -m ${_DNSHOST_mem} \
  -cdrom ${_DNSHOST_ISO} \
  -boot order=cd,menu=on,splash=${_DNS_splash},splash-time=3000 \
```

```
-drive if=none,id=drive0,cache=none,aio=threads,format=raw,file=${_DNSHOST_img} \  
-device virtio-blk,drive=drive0 \  
-netdev tap,id=nd0,ifname=${_DNSHOST_tap7},script=no,downscript=no \  
-device e1000,netdev=nd0,mac=${_DNSHOST_mac1} \  
-netdev tap,id=nd1,ifname=${_DNSHOST_tap8},script=no,downscript=no \  
-device e1000,netdev=nd1,mac=${_DNSHOST_mac2} \  
-netdev tap,id=nd2,ifname=${_DNSHOST_tap11},script=no,downscript=no \  
-device e1000,netdev=nd2,mac=${_DNSHOST_mac3} \  
-name \"${_DNSHOST_name}\" &
```

=====

SCRIPT: external1.sh

```
#!/bin/sh  
# IPFW Primer  
# License: 3-clause BSD  
# Author: Jim Brown, jpb@jimby.name  
# Code: https://github.com/jimmyb-gh/ipfw-primer  
#  
# FreeBSD QEMU VM startup script for external1 VM.  
#  
# external1.sh: FreeBSD QEMU VM startup script for external1 VM.  
# Usage: sudo /bin/sh external1.sh  
# Note: Set up for serial console. Start another session and telnet to the port shown.  
#  
# FreeBSD QEMU VM startup script  
#  
# external1.sh  
#  
#set -x  
  
# pick up environment for this run  
. ./vm_envs.sh  
  
echo [_EXTERNAL1_ISO]  
echo [_EXTERNAL1_mem]  
echo [_EXTERNAL1_hdsz]  
echo [_EXTERNAL1_img]  
echo [_EXTERNAL1_mac]  
echo [_EXTERNAL1_name]  
echo [_EXTERNAL1_tap1]  
echo [_EXTERNAL1_telnetport]  
  
#  
#exit  
#  
  
echo
```

```

echo "NOTE!!! telnet server running! To start QEMU telnet to localhost
${_EXTERNAL1_telnetport}"
echo

/usr/local/bin/qemu-system-x86_64 -monitor none \
  -serial telnet:localhost:${_EXTERNAL1_telnetport},server=on,wait=off \
  -cpu qemu64 \
  -vga cirrus \
  -m ${_EXTERNAL1_mem} \
  -cdrom ${_EXTERNAL1_ISO} \
  -boot order=cd,menu=on,splash=${_EX1_splash},splash-time=3000 \
  -drive if=none,id=drive0,cache=none,aio=threads,format=raw,file=${_EXTERNAL1_img} \
  -device virtio-blk,drive=drive0 \
  -netdev tap,id=nd0,ifname=${_EXTERNAL1_tap1},script=no,downscript=no \
  -device e1000,netdev=nd0,mac=${_EXTERNAL1_mac} \
  -name \"${_EXTERNAL1_name}\" &

```

=====

SCRIPT: external2.sh

```

#!/bin/sh
# IPFW Primer
# License: 3-clause BSD
# Author: Jim Brown, jpb@jimby.name
# Code: https://github.com/jimmyb-gh/ipfw-primer
#
# FreeBSD QEMU VM startup script for external2 VM.
#
# external2.sh: FreeBSD QEMU VM startup script for external2 VM.
# Usage: sudo /bin/sh external2.sh
# Note: Set up for serial console. Start another session and telnet to the port shown.
#
# FreeBSD qemu vm startup script
#
# external2.sh
#
#set -x

# pick up environment for this run
. ./vm_envs.sh

echo [ISO=${_EXTERNAL2_ISO}]
echo [mem=${_EXTERNAL2_mem}]
echo [hdsiz=${_EXTERNAL2_hdsiz}]
echo [img=${_EXTERNAL2_img}]
echo [mac=${_EXTERNAL2_mac}]
echo [name=${_EXTERNAL2_name}]

```

```

echo [tap2=${_EXTERNAL2_tap2}]
echo [telnetport=${_EXTERNAL2_telnetport}]

#
#exit
#

echo
echo "NOTE!!! telnet server running! To start QEMU telnet to localhost
${_EXTERNAL2_telnetport}"
echo

/usr/local/bin/qemu-system-x86_64 -monitor none \
  -serial telnet:localhost:${_EXTERNAL2_telnetport},server=on,wait=off \
  -cpu qemu64 \
  -vga cirrus \
  -m ${_EXTERNAL2_mem} \
  -cdrom ${_EXTERNAL2_ISO} \
  -boot order=cd,menu=on,splash=${_EX2_splash},splash-time=3000 \
  -drive if=none,id=drive0,cache=none,aio=threads,format=raw,file=${_EXTERNAL2_img} \
  -device virtio-blk,drive=drive0 \
  -netdev tap,id=nd0,ifname=${_EXTERNAL2_tap2},script=no,downscript=no \
  -device e1000,netdev=nd0,mac=${_EXTERNAL2_mac} \
  -name \"${_EXTERNAL2_name}\" &

```

=====

SCRIPT: external3.sh

```

#!/bin/sh
# IPFW Primer
# License: 3-clause BSD
# Author: Jim Brown, jpb@jimby.name
# Code: https://github.com/jimmyb-gh/ipfw-primer
#
# FreeBSD QEMU VM startup script for external3 VM.
#
# external3.sh: FreeBSD QEMU VM startup script for external3 VM.
# Usage: sudo /bin/sh external3.sh
# Note: Set up for serial console. Start another session and telnet to the port shown.
#
# FreeBSD QEMU VM startup script
#
# external3.sh
#
#set -x

# pick up environment for this run
. ./vm_envs.sh

```

```

echo [ISO=${_EXTERNAL3_ISO}]
echo [mem=${_EXTERNAL3_mem}]
echo [hdsiz=${_EXTERNAL3_hdsiz}]
echo [img=${_EXTERNAL3_img}]
echo [mac=${_EXTERNAL3_mac}]
echo [name=${_EXTERNAL3_name}]
echo [tap3=${_EXTERNAL3_tap3}]
echo [telnetport=${_EXTERNAL3_telnetport}]

#
#exit
#

echo
echo "NOTE!!! telnet server running! To start QEMU telnet to localhost
${_EXTERNAL3_telnetport}"
echo

/usr/local/bin/qemu-system-x86_64 -monitor none \
  -serial telnet:localhost:${_EXTERNAL3_telnetport},server=on,wait=off \
  -cpu qemu64 \
  -vga cirrus \
  -m ${_EXTERNAL3_mem} \
  -cdrom ${_EXTERNAL3_ISO} \
  -boot order=cd,menu=on,splash=${_EX3_splash},splash-time=3000 \
  -drive if=none,id=drive0,cache=none,aio=threads,format=raw,file=${_EXTERNAL3_img} \
  -device virtio-blk,drive=drive0 \
  -netdev tap,id=nd0,ifname=${_EXTERNAL3_tap3},script=no,downscript=no \
  -device e1000,netdev=nd0,mac=${_EXTERNAL3_mac} \
  -name \"${_EXTERNAL3_name}\" &

```

=====

SCRIPT: firewall.sh

```

#!/bin/sh
# IPFW Primer
# License: 3-clause BSD
# Author: Jim Brown, jpb@jimby.name
# Code: https://github.com/jimmyb-gh/ipfw-primer
#
# FreeBSD QEMU VM startup script for firewall VM.
#
# firewall.sh: FreeBSD QEMU VM startup script for firewall VM.

```

```

# Usage: sudo /bin/sh firewall.sh
# Note: Set up for serial console. Start another session and telnet to the port shown.
#
# FreeBSD QEMU VM startup script
#
# firewall.sh
#
#set -x

# pick up environment for this run
. ./vm_envs.sh

echo [$_FIREWALL_ISO]
echo [$_FIREWALL_mem]
echo [$_FIREWALL_hdsiz]
echo [$_FIREWALL_img]
echo [$_FIREWALL_mac1]
echo [$_FIREWALL_mac2]
echo [$_FIREWALL_name]
echo [$_FIREWALL_tap0]
echo [$_FIREWALL_tap4]
echo [$_FIREWALL_telnetport]

#exit

# Note - the firewall has two interfaces - em0 and em1.
#       em0 is considered the 'external' interface and
#       em1 is considered the 'internal' interface.

echo
echo "NOTE!!! telnet server running! To start QEMU telnet to localhost
$_FIREWALL_telnetport"
echo

/usr/local/bin/qemu-system-x86_64 -monitor none \
  -serial telnet:localhost:${_FIREWALL_telnetport},server=on,wait=off \
  -cpu qemu64 \
  -display gtk \
  -vga cirrus \
  -m ${_FIREWALL_mem} \
  -cdrom ${_FIREWALL_ISO} \
  -boot order=cd,menu=on,splash=${_FW_splash},splash-time=3000 \
  -drive if=none,id=drive0,cache=none,aio=threads,format=raw,file=${_FIREWALL_img} \
  -device virtio-blk,drive=drive0 \
  -netdev tap,id=nd0,ifname=${_FIREWALL_tap0},script=no,downscript=no \
  -device e1000,netdev=nd0,mac=${_FIREWALL_mac1} \
  -netdev tap,id=nd1,ifname=${_FIREWALL_tap4},script=no,downscript=no \
  -device e1000,netdev=nd1,mac=${_FIREWALL_mac2} \
  -name \"${_FIREWALL_name}\" &

```

=====

SCRIPT: firewall2.sh

```
#!/bin/sh
# IPFW Primer
# License: 3-clause BSD
# Author: Jim Brown, jpb@jimby.name
# Code: https://github.com/jimmyb-gh/ipfw-primer
#
# FreeBSD QEMU VM startup script for firewall2 VM.
#
# firewall2.sh: FreeBSD QEMU VM startup script for firewall2 VM.
# Usage: sudo /bin/sh firewall2.sh
# Note: Set up for serial console. Start another session and telnet to the port shown.
#
# FreeBSD QEMU VM startup script
#
# firewall2.sh
#
#set -x

# pick up environment for this run
. ./vm_envs.sh

echo [ISO=${_FIREWALL2_ISO}]
echo [mem=${_FIREWALL2_mem}]
echo [hdsiz=${_FIREWALL2_hdsiz}]
echo [img=${_FIREWALL2_img}]
echo [mac1=${_FIREWALL2_mac1}]
echo [mac2=${_FIREWALL2_mac2}]
echo [name=${_FIREWALL2_name}]
echo [tap9=${_FIREWALL2_tap9}]
echo [tap10=${_FIREWALL2_tap10}]
echo [telnetport=${_FIREWALL2_telnetport}]

#exit

# Note - the firewall has two interfaces - em0 and em1.
#       em0 is considered the 'external' interface and
#       em1 is considered the 'internal' interface.

echo
echo "NOTE!!! telnet server running! To start QEMU telnet to localhost
${_FIREWALL2_telnetport}"
echo

/usr/local/bin/qemu-system-x86_64 -monitor none \
  -serial telnet:localhost:${_FIREWALL2_telnetport},server=on,wait=off \
  -cpu qemu64 \
```



```

-display gtk \
-vga cirrus \
-m ${_FIREWALL2_mem} \
-cdrom ${_FIREWALL2_ISO} \
-boot order=cd,menu=on,splash=${_FW2_splash},splash-time=3000 \
-drive if=none,id=drive0,cache=none,aio=threads,format=raw,file=${_FIREWALL2_img} \
-device virtio-blk,drive=drive0 \
-netdev tap,id=nd0,ifname=${_FIREWALL2_tap9},script=no,downscript=no \
-device e1000,netdev=nd0,mac=${_FIREWALL2_mac1} \
-netdev tap,id=nd1,ifname=${_FIREWALL2_tap10},script=no,downscript=no \
-device e1000,netdev=nd1,mac=${_FIREWALL2_mac2} \
-name \"${_FIREWALL2_name}\" &

```

=====

SCRIPT: internal.sh

```

#!/bin/sh
# IPFW Primer
# License: 3-clause BSD
# Author: Jim Brown, jpb@jimby.name
# Code: https://github.com/jimmyb-gh/ipfw-primer
#
# FreeBSD QEMU VM startup script for internal VM.
#
# internal.sh: FreeBSD QEMU VM startup script for internal VM.
# Usage: sudo /bin/sh internal.sh
# Note: Set up for serial console. Start another session and telnet to the port shown.
#
# FreeBSD QEMU VM startup script
#
# internal.sh
#
#set -x

# pick up environment for this run
. ./vm_envs.sh

echo [ISO=${_INTERNAL_ISO}]
echo [mem=${_INTERNAL_mem}]
echo [hdsiz=${_INTERNAL_hdsiz}]
echo [img=${_INTERNAL_img}]
echo [mac=${_INTERNAL_mac}]
echo [name=${_INTERNAL_name}]
echo [tap5=${_INTERNAL_tap5}]
echo [telnetport=${_INTERNAL_telnetport}]

#

```

```

#exit

echo
echo "NOTE!!! telnet server running! To start QEMU telnet to localhost
${_INTERNAL_telnetport}"
echo

/usr/local/bin/qemu-system-x86_64 -monitor none \
  -serial telnet:localhost:${_INTERNAL_telnetport},server=on,wait=off \
  -cpu qemu64 \
  -vga cirrus \
  -m ${_INTERNAL_mem} \
  -cdrom ${_INTERNAL_ISO} \
  -boot order=cd,menu=on,splash=${_INT_splash},splash-time=3000 \
  -drive if=none,id=drive0,cache=none,aio=threads,format=raw,file=${_INTERNAL_img} \
  -device virtio-blk,drive=drive0 \
  -netdev tap,id=nd0,ifname=${_INTERNAL_tap5},script=no,downscript=no \
  -device e1000,netdev=nd0,mac=${_INTERNAL_mac} \
  -name \"${_INTERNAL_name}\" &

=====

#!/bin/sh
# FreeBSD qemu vm startup script
#
# jail1.sh
#
#set -x

# pick up environment for this run
. ./vm_envs.sh

echo [ISO=${_JAIL1_ISO}]
echo [mem=${_JAIL1_mem}]
echo [hdsz=${_JAIL1_hdsz}]
echo [img=${_JAIL1_img}]
echo [mac=${_JAIL1_mac}]
echo [name=${_JAIL1_name}]
echo [tap2=${_JAIL1_tap2}]
echo [telnetport=${_JAIL1_telnetport}]

#
#exit
#

# minimal check that environment is sane
#if [ "X${_FBSD_ISO}" = "X" -o ! -s ${_FBSD_ISO} ]
#then
# echo "Parameter or file failure on _FBSD_ISO [${_FBSD_ISO}]"

```

```

# echo "Check vm_envs.sh"
# exit 1
#fi
#
#

#echo
#echo "NOTE!!! telnet server running! To start QEMU telnet to localhost
$_JAIL1_telnetport"
#echo
# -serial telnet:localhost:${_JAIL1_telnetport},server=on,wait=on \

/usr/local/bin/qemu-system-x86_64 -monitor none \
  -serial telnet:localhost:${_JAIL1_telnetport},server=on,wait=off \
  -cpu qemu64 \
  -vga cirrus \
  -m ${_JAIL1_mem} \
  -cdrom ${_JAIL1_ISO} \
  -boot order=cd,menu=on,splash=${_JAIL1_splash},splash-time=3000 \
  -drive if=none,id=drive0,cache=none,aio=threads,format=raw,file=${_JAIL1_img} \
  -device virtio-blk,drive=drive0 \
  -netdev tap,id=nd0,ifname=${_JAIL1_tap12},script=no,downscript=no \
  -device e1000,netdev=nd0,mac=${_JAIL1_mac} \
  -name \"${_JAIL1_name}\" &

```

=====

SCRIPT: mkbr.sh

```

#!/bin/sh
# IPFW Primer
# License: 3-clause BSD
# Author: Jim Brown, jpb@jimby.name
# Code: https://github.com/jimmyb-gh/ipfw-primer
#
# FreeBSD startup script for bridge and tap devices.
#
# mkbr.sh: FreeBSD startup script for bridge and tap devices.
# EXAMPLE Usage: sudo /bin/sh mkbr.sh reset bridge0 tap0 tap1 bridge1 tap2 bridge2
tap3 tap4 tap5 em0
#
# mkbr.sh - manage bridge and tap interfaces for FreeBSD.
#
# Have fun, but don't blame me if it smokes your machine.
#
# This script is used to start the bridge and tap interfaces.
#
# To create one bridge, two tap interfaces, and connect the

```

```

# local ethernet interace (here em0), run under sudo as follows:
# sudo /bin/sh mkbr.sh reset bridge0 tap0 tap1 em0
#
# The script can be used to create any number of bridges and taps
# for any internal network design:
# sudo /bin/sh mkbr.sh reset bridge0 tap0 tap1 bridge1 tap2 tap3 tap4 bridge2 tap5
em0 ... etc.
#
# To add other taps to existing bridges, do not specify the "reset" parameter.
# sudo /bin/sh mkbr.sh bridge0 tap10 tap11 bridge1 tap12 tap13 ... etc.
#
# To delete all bridge and tap devices:
# sudo /bin/sh mkbr.sh reset
#
#

#set -x

usage() {
    echo "Usage: mkbr.sh ["reset"] <bridgeN> <tapA> [[<bringeM>] <tapB> <tapC> ...]"
    echo "You must be root to run this script."
    exit 1
}

if [ "$0" != "X`id -u`" ]
then
    usage
fi

if [ $# = 0 ]
then
    usage;
fi

if [ $1 = "reset" ]
then
    echo
    echo "Note - if_bridge and/or if_tap may be compiled into the kernel and can't be
unloaded. Adjust interfaces manually if necessary."
    echo
    echo "unloading..."
    kldunload if_bridge
    kldunload if_tap
    echo
    echo "Deleting any remaining bridge and tap devices:"

    for i in ifconfig -l
    do
        echo "Interface: ${i}"
    done
fi

```

```

case ${i} in

    bridge*)
        echo " ... destroying bridge ${i}"
        ifconfig ${i} destroy
        ;;
    tap*)
        echo " ... destroying tap ${i}"
        ifconfig ${i} destroy
        ;;
esac
done

sleep 1
echo "loading..."
kldload if_bridge
kldload if_tap
shift
RESET="Y"
echo "RESET=Y"
# Before using the tap devices in QEMU, two sysctls require adjustment:
sysctl net.link.tap.user_open=1
sysctl net.link.tap.up_on_open=1
else
    RESET="N"
    echo "RESET=N"
fi

PARAM=$1

while [ "X${PARAM}" != "X" ]
do
# echo "PARAM=[${PARAM}]"

case $PARAM in

    bridge*) BRIDGE=$1

        # if [ "$RESET" = "Y" ]
        # then
            echo ifconfig $BRIDGE create
            ifconfig $BRIDGE create
            echo ifconfig $BRIDGE
            ifconfig $BRIDGE
        # fi
        echo ifconfig $BRIDGE up
            ifconfig $BRIDGE up
        ;;

```

```

tap*) TAP=$1
# if [ "$RESET" = "Y" ]
# then
    echo ifconfig $TAP create
    ifconfig $TAP create
# fi
echo "ifconfig $BRIDGE addm $TAP "
    ifconfig $BRIDGE addm $TAP
;;

*) echo "*** Checking to see if $1 is a valid interface"
TMPINT=$1
RESULT="IS NOT"
for i in ifconfig -l
do
#     echo $i
    if [ "${i}X" = "${TMPINT}X" ]
    then
        echo "Found a valid interface: ${TMPINT} Adding it to the bridge. Check
results."
        echo "ifconfig $BRIDGE addm $TMPINT"
        ifconfig $BRIDGE addm $TMPINT
        RESULT="IS"
        break;
    else
        echo -n "."
    fi
done

    echo "Interface ${TMPINT} $RESULT a valid interface."
;;
esac

shift
PARAM=$1
done

exit 0

=====

SCRIPT: runvm.sh

#!/bin/sh
# IPFW Primer
# License: 3-clause BSD
# Author: Jim Brown, jpb@jimby.name
# Code: https://github.com/jimmyb-gh/ipfw-primer
#

```

```

# FreeBSD QEMU VM startup script for multiple VMs at once.
#
# runvm.sh: FreeBSD QEMU VM startup script for multiple VMs.
# EXAMPLE Usage: /bin/sh runvm.sh firewall external1 external2 internal
#
# location: FreeBSD Host
#
# runvm.sh - run virtual machines specified on the command line.
#
# To use this script, run mkbr.sh first to set up the bridge and
# tap configurations for the desired network architecture.
#
# NOTE: this script works best on XFCE4 desktop as it takes advantage of the
#       xfce4-terminal and it's ability to use multiple tabs.
#
# >>>> It is unlikely to work on another desktop. <<<<
#
# Essentially, this script is a big case statement. It gets the
# command line names of the virtual machines and calls a function
# that starts the virtual machine.
#

# pick up environment for this run
. ./vm_envs.sh

#set -x

#WKDIR=$HOME/LAB/SCRIPTS
export WKDIR=$HOME/ipfw

echo "[${WKDIR}]"

usage() {
    echo "Usage: /bin/sh runvm.sh vmname [vmname ...]"
    echo "Each virtual machine opens up on xfce4-terminal with two tabs -"
    echo "  one for the qemu virtual machine, and one for the serial"
    echo "  terminal interface."
    echo ""
    exit 1
}

CURDIR=pwd

if [ "${CURDIR}" != "${WKDIR}/SCRIPTS" ]
then
    usage;
fi

```

```

if [ $# = 0 ]
then
    usage;
fi

# Functions for each VM

dnshost_vm () {
    # DNS host
    echo "in function: [${_DNSHOST_telnetport}]"
    xfce4-terminal --window --geometry="80x24+50+50" --zoom="-1" \
        -T "${_DNSHOST_name}" -e "bash -c \"cd ${WKDIR}/SCRIPTS && sudo /bin/sh
dnshost.sh ; bash\"" \
        --tab -T "${_DNSHOST_name}" -e "bash -c \"cd ${WKDIR}/SCRIPTS && sleep 2 &&
(. ./vm_envs.sh;telnet -4 localhost ${_DNSHOST_telnetport}); bash\""
    return
}

external1_vm () {
    # external1
    echo "in function: [${_EXTERNAL1_telnetport}]"
    xfce4-terminal --window --geometry="80x24+75+75" --zoom="-1" \
        -T "${_EXTERNAL1_name}" -e "bash -c \"cd $WKDIR/SCRIPTS && sudo /bin/sh
external1.sh ; bash\"" \
        --tab -T "${_EXTERNAL1_name}" -e "bash -c \"cd $WKDIR/SCRIPTS && sleep 2 &&
(. ./vm_envs.sh;telnet localhost ${_EXTERNAL1_telnetport}); bash\""
    return
}

external2_vm () {
    # external2
    echo "in function: [${_EXTERNAL2_telnetport}]"
    xfce4-terminal --window --geometry="80x24+100+100" --zoom="-1" \
        -T "${_EXTERNAL2_name}" -e "bash -c \"cd $WKDIR/SCRIPTS && sudo /bin/sh
external2.sh ; bash\"" \
        --tab -T "${_EXTERNAL2_name}" -e "bash -c \"cd $WKDIR/SCRIPTS && sleep 2 &&
(. ./vm_envs.sh;telnet localhost ${_EXTERNAL2_telnetport}); bash\""
    return
}

external3_vm () {
    # external3
    echo "in function: [${_EXTERNAL3_telnetport}]"
    xfce4-terminal --window --geometry="80x24+125+125" --zoom="-1" \
        -T "${_EXTERNAL3_name}" -e "bash -c \"cd $WKDIR/SCRIPTS && sudo /bin/sh
external3.sh ; bash\"" \
        --tab -T "${_EXTERNAL3_name}" -e "bash -c \"cd $WKDIR/SCRIPTS && sleep 2 &&
(. ./vm_envs.sh;telnet localhost ${_EXTERNAL3_telnetport}); bash\""
    return
}

```



```

}

firewall_vm () {
# Firewall
echo "in function: [${_FIREWALL_telnetport}]"
xfce4-terminal --window --geometry="80x24+150+150" --zoom="-1" \
-T "${_FIREWALL_name}" -e "bash -c \"cd $WKDIR/SCRIPTS && sudo /bin/sh
firewall.sh ; bash\"" \
--tab -T "${_FIREWALL_name}" -e "bash -c \"cd $WKDIR/SCRIPTS && sleep 2 && (.
./vm_envs.sh; telnet localhost ${_FIREWALL_telnetport}); bash\""
return
}

firewall2_vm () {
# Firewall2
echo "in function: [${_FIREWALL2_telnetport}]"
xfce4-terminal --window --geometry="80x24+175+175" --zoom="-1" \
-T "${_FIREWALL2_name}" -e "bash -c \"cd $WKDIR/SCRIPTS && sudo /bin/sh
firewall2.sh ; bash\"" \
--tab -T "${_FIREWALL2_name}" -e "bash -c \"cd $WKDIR/SCRIPTS && sleep 2 &&
(. ./vm_envs.sh;telnet localhost ${_FIREWALL2_telnetport}); bash\""
return
}

internal_vm () {
# internal
echo "in function: [${_INTERNAL_telnetport}]"
xfce4-terminal --window --geometry="80x24+200+200" --zoom="-1" \
-T "${_INTERNAL_name}" -e "bash -c \"cd $WKDIR/SCRIPTS && sudo /bin/sh
internal.sh ; bash\"" \
--tab -T "${_INTERNAL_name}" -e "bash -c \"cd $WKDIR/SCRIPTS && sleep 2 && (.
./vm_envs.sh;telnet localhost ${_INTERNAL_telnetport}); bash\""
return
}

v6only_vm () {
# v6only
echo "in function: [${_V6ONLY_telnetport}]"
xfce4-terminal --window --geometry="80x24+225+225" --zoom="-1" \
-T "${_V6ONLY_name}" -e "bash -c \"cd $WKDIR/SCRIPTS && sudo /bin/sh
v6only.sh ; bash\"" \
--tab -T "${_V6ONLY_name}" -e "bash -c \"cd $WKDIR/SCRIPTS && sleep 2 && (.
./vm_envs.sh;telnet localhost ${_V6ONLY_telnetport}); bash\""
return
}

#
# Startup the requested VMs
#

```

```

PARAM=$1

while [ "X${PARAM}" != "X" ]
do
    echo "PARAM = [${PARAM}]"

    case ${PARAM} in

        dnshost)
            echo "dnshost ..."
            echo "_DNSHOST_telnetport = [${_DNSHOST_telnetport}]"
            dnshost_vm
            ;;

        external1)
            echo "external1 ..."
            echo "_EXTERNAL1_telnetport = [${_EXTERNAL1_telnetport}]"
            external1_vm
            ;;

        external2)
            echo "external2 ..."
            echo "_EXTERNAL2_telnetport = [${_EXTERNAL2_telnetport}]"
            external2_vm
            ;;

        external3)
            echo "external3 ..."
            echo "_EXTERNAL3_telnetport = [${_EXTERNAL3_telnetport}]"
            external3_vm
            ;;

        firewall)
            echo "firewall ..."
            echo "_FIREWALL_telnetport = [${_FIREWALL_telnetport}]"
            firewall_vm
            ;;

        firewall2)
            echo "firewall2 ..."
            echo "_FIREWALL2_telnetport = [${_FIREWALL2_telnetport}]"
            firewall2_vm
            ;;

        internal)
            echo "internal ..."
            echo "_INTERNAL_telnetport = [${_INTERNAL_telnetport}]"
            internal_vm
            ;;
    esac
done

```

```

v6only)
    echo "v6only ..."
    echo "_V6ONLY_telnetport = [${_V6ONLY_telnetport}]"
    v6only_vm
    ;;

*)
    echo ""
    echo "*** ERROR: NO VM NAMED [$PARAM]"
    echo ""
    ;;

esac

shift

sleep 3

PARAM=$1
done

exit 0

```

=====

SCRIPT: swim.sh

```

#!/bin/sh
# IPFW Primer
# License: 3-clause BSD
# Author: Jim Brown, jpb@jimby.name
# Code: https://github.com/jimmyb-gh/ipfw-primer
#
# Serial Window Management Script Using tmux. (swim.sh)
#
# Usage: /bin/sh swim.sh
# Note: This program manages multiple serial terminal windows for QEMU
#       VMs on the host.
#       Make sure to uncomment the lines below for the windows you want.
#set -x

# Check for an existing tmux session
tmux has-session -t 0 2>/dev/null

if [ $? != 0 ]; then
    tmux new-session -d -s 0

    tmux new-window -t 0:1 -n 'firewall' 'echo; echo Use \"telnet localhost 4450\" for
    firewall ; echo; /bin/sh'

```

```

tmux new-window -t 0:2 -n 'external1' 'echo; echo Use \"telnet localhost 4410\" for
external1; echo; /bin/sh'
tmux new-window -t 0:3 -n 'external2' 'echo; echo Use \"telnet localhost 4420\" for
external2; echo; /bin/sh'
tmux new-window -t 0:4 -n 'external3' 'echo; echo Use \"telnet localhost 4430\" for
external3; echo; /bin/sh'
# tmux new-window -t 0:5 -n 'internal' 'echo; echo Use \"telnet localhost 44200\"
for internal; echo; /bin/sh'
# tmux new-window -t 0:6 -n 'firewall2' 'echo; echo Use \"telnet localhost 4250\"
for firewall2; echo; /bin/sh'
# tmux new-window -t 0:7 -n 'v6only' 'echo; echo Use \"telnet localhost 4460\"
for v6only; echo; /bin/sh'
# tmux new-window -t 0:8 -n 'dnshost' 'echo; echo Use \"telnet localhost 4453\"
for dnshost; echo; /bin/sh'
# tmux new-window -t 0:9 -n 'jail1' 'echo; echo Use \"telnet localhost 4470\"
for jail1; echo; /bin/sh'

# Set the default command shell
set-option -g default-command "/bin/sh"
fi

# Set the focus on window 0:0, your existing shell.
tmux select-window -t 0:0

# Attach to the session
tmux attach-session -t 0

exit

```

=====

SCRIPT: scim.sh

```

#!/bin/sh
# IPFW Primer
# License: 3-clause BSD
# Author: Jim Brown, jpb@jimby.name
# Code: https://github.com/jimmyb-gh/ipfw-primer
#
# Serial Window Management Script Using screen. (scim.sh)
#
# Usage: /bin/sh scim.sh
# Note: This program manages multiple serial terminal windows for QEMU VMs.
#       Make sure to uncomment the lines below for the windows you want.
#
# Note:
# In order to show the status line for the list of active windows,
# this program requires a .screenrc file in the $HOME directory with
# the following directives:

```

```

#
# hardstatus alwayslastline
# hardstatus string "%{= bw}%-w%{= rW}[%n %t]%-}%+w %=%{= kW} %H | %Y-%m-%d %c"
#

screen -list 2> /dev/null

# Check if the session is already live
if [ $? != 0 ]; then

    # Create a new session and add windows
    screen -dmS newsession          # Start a new session

    # Uncomment windows as needed.
    screen -S newsession -X screen -t "firewall" sh -c "echo; echo Use \"telnet
localhost 4450\" for firewall ; echo; /bin/sh"
    screen -S newsession -X screen -t "external1" sh -c "echo; echo Use \"telnet
localhost 4410\" for external1 ; echo; /bin/sh"
    screen -S newsession -X screen -t "external2" sh -c "echo; echo Use \"telnet
localhost 4420\" for external1 ; echo; /bin/sh"
    screen -S newsession -X screen -t "external3" sh -c "echo; echo Use \"telnet
localhost 4430\" for external1 ; echo; /bin/sh"
    # screen -S newsession -X screen -t "internal" sh -c "echo; echo Use \"telnet
localhost 44200\" for external1 ; echo; /bin/sh"
    # screen -S newsession -X screen -t "firewall2" sh -c "echo; echo Use \"telnet
localhost 4250\" for external1 ; echo; /bin/sh"
    # screen -S newsession -X screen -t "v6only" sh -c "echo; echo Use \"telnet
localhost 4460\" for external1 ; echo; /bin/sh"
    # screen -S newsession -X screen -t "dnshost" sh -c "echo; echo Use \"telnet
localhost 4453\" for external1 ; echo; /bin/sh"
    # screen -S newsession -X screen -t "jail1" sh -c "echo; echo Use \"telnet
localhost 4470\" for external1 ; echo; /bin/sh"

# Focus on window 0
screen -S newsession -X select 0
fi

# Light it up.
screen -x newsession -p 0

=====

SCRIPT: v6only.sh

#!/bin/sh
# IPFW Primer
# License: 3-clause BSD
# Author: Jim Brown, jpb@jimby.name
# Code: https://github.com/jimmyb-gh/ipfw-primer

```

```

#
# FreeBSD QEMU VM startup script for v6only VM.
#
# v6only.sh: FreeBSD QEMU VM startup script for v6only VM.
# Usage: sudo /bin/sh v6only.sh
# Note: Set up for serial console. Start another session and telnet to the port shown.
#
# FreeBSD QEMU VM startup script
#
# v6only.sh
#
#set -x

# pick up environment for this run
. ./vm_envs.sh

echo [ISO=${_V6ONLY_ISO}]
echo [mem=${_V6ONLY_mem}]
echo [hdsiz=${_V6ONLY_hdsiz}]
echo [img=${_V6ONLY_img}]
echo [mac=${_V6ONLY_mac}]
echo [name=${_V6ONLY_name}]
echo [tap6=${_V6ONLY_tap6}]
echo [telnetport=${_V6ONLY_telnetport}]

#
#exit

echo
echo "NOTE!!! telnet server running! To start QEMU telnet to localhost
${_V6ONLY_telnetport}"
echo

/usr/local/bin/qemu-system-x86_64 -monitor none \
  -serial telnet:localhost:${_V6ONLY_telnetport},server=on,wait=off \
  -cpu qemu64 \
  -vga cirrus \
  -m ${_V6ONLY_mem} \
  -cdrom ${_V6ONLY_ISO} \
  -boot order=cd,menu=on,splash=${_V6_splash},splash-time=3000 \
  -drive if=none,id=drive0,cache=none,aio=threads,format=raw,file=${_V6ONLY_img} \
  -device virtio-blk,drive=drive0 \
  -netdev tap,id=nd0,ifname=${_V6ONLY_tap6},script=no,downscript=no \
  -device e1000,netdev=nd0,mac=${_V6ONLY_mac} \
  -name \"${_V6ONLY_name}\" &

```

=====

SCRIPT: vm_envs.sh

```
# IPFW Primer
# License: 3-clause BSD
# Author: Jim Brown, jpb@jimby.name
# Code: https://github.com/jimmyb-gh/ipfw-primer
#
# FreeBSD QEMU VM environment script.
#
# vmenv.sh: FreeBSD QEMU VM environment setup script.
# Usage: ./bin/sh vmenv.sh
#
# vm_envs.sh - environment for setting up virtual machines
#             for the IPFW examples lab.
#
# Set the environment variables below (or keep the defaults)
# Note that the default disk size for each virtual machine is
# 4GB - so all five VMs will take up about 32GB if you preallocate
# space.
#
# In brief:
#
# Install FreeBSD on the host machine and update to latest patch level.
# Install desktop software.
# Install QEMU (latest)
# Install nmap (needed for ncat)
# Install sudo
#
#
# The script mkbr.sh should be run before starting
# the virtual machines. mkbr.sh sets up the bridge and tap
# devices needed by the VMs.
#
# sudo /bin/sh ./mkbr.sh reset bridge0 tap0 tap1 tap2 tap3 em0 bridge1 tap4 tap5
#
# This will set up the devices needed by QEMU.
#
#
#The file directory layout for the examples is:
#
#   ~/ipfw
#       /SCRIPTS
#           _CreateAllVMs.sh  (create Qemu disks images)
#           dnshost.sh        (run script for dns server VM)
#           external1.sh      (run scripts for external VMs)
#           external2.sh      "
#           external3.sh      "
#           firewall.sh       (run script for firewall VM)
#           firewall2.sh      (run script for firewall2 VM)
#           internal.sh        (script to setup internal host)
#           jail1.sh           (script to setup jail1 host)
```

```

#         v6only.sh           (run script for IPv6 only VM)
#         mkbr.sh            (script to create bridge and tap devices)
#         vm_envs.sh         (script to manage all parameters)
#         runvm.sh           (script to manage all virtual machines)
#
#     /BMP
#         dns_splash_640x480.bmp
#         external1_splash_640x480.bmp
#         external2_splash_640x480.bmp
#         external3_splash_640x480.bmp
#         internal_splash_640x480.bmp
#         jail1_splash_640x480.bmp
#         ipfw2_splash_640x480.bmp
#         ipfw_splash_640x480.bmp
#         v6only_splash_640x480.bmp
#         dnshost_splash_640x480.bmp
#
#     /ISO
#         fbsd.iso            (latest FreeBSD install iso)
#
#     /VM
#         dnshost.qcow2       (Qemu disk image for dns host)
#         external1.qcow2     (Qemu disk image for external hosts)
#         external2.qcow2     "
#         external3.qcow2     "
#         firewall.qcow2      (Qemu disk image for firewall)
#         firewall2.qcow2     (Qemu disk image for firewall2)
#         internal.qcow2      (Qemu disk image for an internal host)
#         jail1.qcow2         (Qemu disk image for an jail1 host)
#         v6only.qcow2        (Qemu disk image for an ipv6only host)
#
#
# Start the VMs and install / test one at a time.
#
#     sudo /bin/sh firewall.sh
#     sudo /bin/sh firewall2.sh
#     sudo /bin/sh external1.sh
#     sudo /bin/sh external2.sh
#     sudo /bin/sh external3.sh
#     sudo /bin/sh internal.sh
#     sudo /bin/sh jail1.sh
#     sudo /bin/sh v6only.sh
#     sudo /bin/sh dnshost.sh
#
# Each install should first utilize DHCP to get a valid IP address
# After install, proceed to update FreeBSD with "freebsd-update fetch install"
# Install packages:
# Use whatever shell you prefer. Bash is listed below.
# Firewall - pkg install bash cmdwatch lynx iperf3 nmap hping3 nginx
# All others - pkg install bash cmdwatch lynx iperf3 nmap hping3 nginx
# DNS host - pkg install bind918 dual-dhclient bash cmdwatch lynx nginx
#
# Reset all IP addresses for static usage:
#

```



```

# Host interface: add 172.16.10.100/24 alias
# Disable any firewall (pf, ipfw, etc.) on the host.
#     BE SURE this is Ok for your environment.
#
# Firewall em0 172.16.10.50/24, default gateway 172.16.10.100
#     em1 10.10.10.50/24
#
# Firewall2 em0 as needed
#     em1 as needed
#
# External1: em0 172.16.10.10/24, default gateway 172.16.10.100
# External2: em0 172.16.10.20/24, default gateway 172.16.10.100
# External3: em0 172.16.10.30/24, default gateway 172.16.10.100
# Internal:  em0 10.10.10.200/24, default gateway 10.10.10.50
#
# v6only  as needed
# dnshost as needed
# jail1  as needed
#
#

export _BASE=/home/jpb/ipfw

# Bridge and tap info
export _FIREWALL_tap0=tap0
export _EXTERNAL1_tap1=tap1
export _EXTERNAL2_tap2=tap2
export _EXTERNAL3_tap3=tap3
export _FIREWALL_tap4=tap4
export _INTERNAL_tap5=tap5
export _JAIL1_tap12=tap12
export _V6ONLY_tap6=tap6
export _DNSHOST_tap7=tap7
export _DNSHOST_tap8=tap8
export _FIREWALL2_tap9=tap9
export _FIREWALL2_tap10=tap10
export _DNSHOST_tap11=tap11

export _bridge0_=bridge0
export bridge1=bridge1
export bridge2=bridge2

# Disk sizes
export _EXTERNAL1_hdsizesize=4G
export _EXTERNAL2_hdsizesize=4G
export _EXTERNAL3_hdsizesize=4G
export _FIREWALL_hdsizesize=4G
export _FIREWALL2_hdsizesize=4G
export _INTERNAL_hdsizesize=4G
export _JAIL1_hdsizesize=8G          # Note larger size disk

```

```

export _V6ONLY_hdsz=4G
export _DNSHOST_hdsz=4G

# Is this needed anymore?
export _FBSD_ISO=${_BASE}/ISO/fbsd.iso

# Boot iso locations
export _DNSHOST_ISO=${_BASE}/ISO/fbsd.iso
export _EXTERNAL1_ISO=${_BASE}/ISO/fbsd.iso
export _EXTERNAL2_ISO=${_BASE}/ISO/fbsd.iso
export _EXTERNAL3_ISO=${_BASE}/ISO/fbsd.iso
export _FIREWALL_ISO=${_BASE}/ISO/fbsd.iso
export _FIREWALL2_ISO=${_BASE}/ISO/fbsd.iso
export _INTERNAL_ISO=${_BASE}/ISO/fbsd.iso
export _JAIL1_ISO=${_BASE}/ISO/fbsd.iso
export _V6ONLY_ISO=${_BASE}/ISO/fbsd.iso

# Memory sizes
export _DNSHOST_mem=1024
export _EXTERNAL1_mem=1024      # lower all to 512 if necessary
export _EXTERNAL2_mem=1024
export _EXTERNAL3_mem=1024
export _FIREWALL_mem=1024
export _FIREWALL2_mem=1024
export _INTERNAL_mem=1024
export _JAIL1_mem=8192         # Note larger size memory for using ZFS
export _V6ONLY_mem=1024

# Qemu disk image locations.
export _DNSHOST_img=${_BASE}/VM/dnshost.qcow2
export _EXTERNAL1_img=${_BASE}/VM/external1.qcow2
export _EXTERNAL2_img=${_BASE}/VM/external2.qcow2
export _EXTERNAL3_img=${_BASE}/VM/external3.qcow2
export _FIREWALL_img=${_BASE}/VM/firewall.qcow2
export _FIREWALL2_img=${_BASE}/VM/firewall2.qcow2
export _INTERNAL_img=${_BASE}/VM/internal.qcow2
export _JAIL1_img=${_BASE}/VM/jail1.qcow2
export _V6ONLY_img=${_BASE}/VM/v6only.qcow2

# MAC addresses
export _DNSHOST_mac1=02:49:53:53:53:53
export _DNSHOST_mac2=02:49:53:53:54:54
export _DNSHOST_mac3=02:49:53:53:55:55
export _EXTERNAL1_mac=02:45:58:54:31:10
export _EXTERNAL2_mac=02:45:58:54:32:20
export _EXTERNAL3_mac=02:45:58:54:33:30
export _FIREWALL_mac1=02:49:50:46:57:41
export _FIREWALL2_mac1=02:49:50:00:22:22
export _FIREWALL_mac2=02:49:50:46:57:42
export _FIREWALL2_mac2=02:49:50:22:22:22

```

```

export _INTERNAL_mac=02:49:4E:54:0a:42
export _JAIL1_mac=02:49:ba:ad:ba:be
export _V6ONLY_mac=02:49:de:ad:be:ef

# VM names
export _DNSHOST_name=DNSHOST
export _EXTERNAL1_name=EXTERNAL1
export _EXTERNAL2_name=EXTERNAL2
export _EXTERNAL3_name=EXTERNAL3
export _FIREWALL_name=FIREWALL
export _FIREWALL2_name=FIREWALL2
export _INTERNAL_name=INTERNAL
export _JAIL1_name=JAIL1
export _V6ONLY_name=V6ONLY

# Slash images
export _DNS_splash=${_BASE}/BMP/dns_splash_640x480.bmp
export _EX1_splash=${_BASE}/BMP/external1_splash_640x480.bmp
export _EX2_splash=${_BASE}/BMP/external2_splash_640x480.bmp
export _EX3_splash=${_BASE}/BMP/external3_splash_640x480.bmp
export _FW_splash=${_BASE}/BMP/ipfw_splash_640x480.bmp
export _FW2_splash=${_BASE}/BMP/ipfw2_splash_640x480.bmp
export _INT_splash=${_BASE}/BMP/internal_splash_640x480.bmp
export _JAIL1_splash=${_BASE}/BMP/jail1_splash_640x480.bmp
export _V6_splash=${_BASE}/BMP/ipv6_splash_640x480.bmp

#
# Telnet ports
export _DNSHOST_telnetport=4453
export _EXTERNAL1_telnetport=4410
export _EXTERNAL2_telnetport=4420
export _EXTERNAL3_telnetport=4430
export _FIREWALL_telnetport=4450
export _FIREWALL2_telnetport=4250
export _INTERNAL_telnetport=44200
export _V6ONLY_telnetport=4460
export _JAIL1_telnetport=4470

# Bridge and Tap configurations.
#
# Note: em0 is used for the host interface.
#       Change as needed.
#
# Two bridge configuration
# Standard examples
#
#           em0
#           |
# External1(tap1) -----bridge0----- (tap0)Firewall
# External2(tap2) -----+ | (tap4)

```

```

# External3(tap3) -----+
#                               |
#                               bridge1
#                               |
# Internal(tap5) -----+
#
# sudo /bin/sh mkbr.sh reset bridge0 tap0 tap1 tap2 tap3 em0
#
#
#
# Two bridge configuration
# NAT & LSNAT examples
#
#
#                               (firewall does LSNAT load balancing)
# External1(tap1) -----bridge0------(tap0)Firewall
# External2(tap2) -----+ |                               (tap4)
# External3(tap3) -----+ |                               |
# (these function as internal machines)   bridge1----em0
#                                           |
# Internal(tap5) -----+
# (this functions as an external machine)
#
# sudo /bin/sh mkbr.sh reset bridge0 tap0 tap1 tap2 tap3 bridge1 tap4 tap5 em0
#
#
#
# Two bridge configuration
# NAT64/DNS64 example
#
#                               ipv4 only       NAT64 Translator
# External1(tap1) -----bridge0------(tap0)Firewall
# (ipv4 only)           +                               (tap4)
# (webserver)           |                               +
#                       |                               |
#                       dnshost(tap7)                 |
#                       (DNS server)                 |
#                       (running DNS64)              |
#                       (tap8)                       |
#                       |                             |
#                       +                             |
#                       ipv6 only                    |
# v6only(tap6) -----bridge1-----+
# (v6 only host)
#
# sudo /bin/sh mkbr.sh reset bridge0 tap0 tap1 tap7 bridge1 tap4 tap6 tap8
#
#

```

```
=====
```

CODE: divert.c

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <netinet/tcp.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <string.h>
#include <err.h>
#include <sys/system.h>

#define DIVERT_PORT 700

void hexdump(void *ptr, int length, const char *hdr, int flags);

int
main(int argc, char *argv[])
{
    int fd, s;
    struct sockaddr_in sin;
    socklen_t sin_len;

    printf("Opening divert on port %d\n",DIVERT_PORT);

    fd = socket(PF_DIVERT, SOCK_RAW, 0);
    if (fd == -1)
        err(1, "socket");

    memset(&sin, 0, sizeof(sin));
    sin.sin_family = AF_INET;
    sin.sin_port = htons(DIVERT_PORT);
    sin.sin_addr.s_addr = 0;

    sin_len = sizeof(struct sockaddr_in);

    s = bind(fd, (struct sockaddr *) &sin, sin_len);
    if (s == -1)
        err(1, "bind");

    for (;;) {
        ssize_t n;
        char packet[IP_MAXPACKET];
        struct ip *ip;
        struct tcphdr *th;
        int hlen;
        char src[64], dst[64], printbuff[12];
```

```

memset(src, 0, sizeof(src));
memset(dst, 0, sizeof(dst));
memset(printbuff, 0, sizeof(printbuff));

memset(packet, 0, sizeof(packet));
n = recvfrom(fd, packet, sizeof(packet), 0,
    (struct sockaddr *) &sin, &sin_len);
if (n == -1) {
    warn("recvfrom");
    continue;
}
if (n < sizeof(struct ip)) {
    warnx("packet is too short");
    continue;
}

ip = (struct ip *) packet;
hlen = ip->ip_hl << 2;
if (hlen < sizeof(struct ip) || ntohs(ip->ip_len) < hlen ||
    n < ntohs(ip->ip_len)) {
    warnx("invalid IPv4 packet");
    continue;
}

th = (struct tcphdr *) (packet + hlen);

if (inet_ntop(AF_INET, &ip->ip_src, src,
    sizeof(src)) == NULL)
    (void)strlcpy(src, "?", sizeof(src));

if (inet_ntop(AF_INET, &ip->ip_dst, dst,
    sizeof(dst)) == NULL)
    (void)strlcpy(dst, "?", sizeof(dst));

printf("%s:%u -> %s:%u\n",
    src,
    ntohs(th->th_sport),
    dst,
    ntohs(th->th_dport)
);

/*
 * dump the packet in hex and ascii with hexdump(3)
 */

hexdump((void *)packet, n, "|", 0);

n = sendto(fd, packet, n, 0, (struct sockaddr *) &sin,

```

```
        sin_len);
    if (n == -1)
        warn("sendto");
}

return 0;
}
```

Appendix C: Appendix C: Networking References

References for understanding IP based communications and building firewalls.

From /etc/rc.firewall:

Building Internet Firewalls, 2nd Edition
Brent Chapman and Elizabeth Zwicky

O'Reilly & Associates, Inc
ISBN 1-56592-871-7
<http://www.ora.com/>
<http://www.oreilly.com/catalog/fire2/>

For a more advanced treatment of Internet Security read:

Firewalls and Internet Security: Repelling the Wily Hacker, 2nd Edition
William R. Cheswick, Steven M. Bellowin, Aviel D. Rubin

Addison-Wesley / Prentice Hall
ISBN 0-201-63466-X
<http://www.pearsonhighered.com/>
<http://www.pearsonhighered.com/educator/academic/product/0,3110,020163466X,00.html>

Additional references:

TCP/IP Illustrated, Volume 1: The Protocols
Author: W. Richard Stevens
Publisher: Addison-Wesley Professional
Publisher Website: Addison-Wesley Professional
Date Published: November 1994
ISBN: 978-0201633467

The TCP/IP Guide: A Comprehensive, Illustrated Internet Protocols Reference
Author: Charles M. Kozierok
Publisher: No Starch Press
Publisher Website: No Starch Press
Date Published: October 2005
ISBN: 978-1593270476

Internetworking with TCP/IP Volume One: Principles, Protocols, and Architecture
Author: Douglas E. Comer
Publisher: Pearson
Publisher Website: Pearson
Date Published: 6th Edition, 2013
ISBN: 978-0136085300

Computer Networks: A Systems Approach
Authors: Larry L. Peterson and Bruce S. Davie
Publisher: Morgan Kaufmann
Publisher Website: Morgan Kaufmann
Date Published: 6th Edition, 2021
ISBN: 978-0128182000

Additional resources regarding firewalls:

Network Security, Firewalls, and VPNs
Authors: J. Michael Stewart, Denise Kinsey
Publisher: Jones & Bartlett Learning
Publisher Website: Jones & Bartlett Learning
Date Published: October 2020
ISBN: 978-1284183696

Firewall Fundamentals
Author: David W. Chapman Jr.
Publisher: Cisco Press
Publisher Website: Cisco Press
Date Published: June 2006
ISBN: 978-1587052213

The Best Damn Firewall Book Period
Author: Thomas W. Shinder
Publisher: Syngress
Publisher Website: Syngress
Date Published: January 2008
ISBN: 978-1597492188

Guide to Firewalls and Network Security: Intrusion Detection and VPNs
Authors: Michael E. Whitman, Herbert J. Mattord, Richard Austin, Greg Holden
Publisher: Cengage Learning
Publisher Website: Cengage Learning
Date Published: 2008
ISBN: 978-1435420168

Appendix D: Appendix D. Managing Serial Terminals with **tmux** and **screen**

D.1. Using **tmux(1)** for Managing Serial Terminals

Install **tmux** with:

```
# pkg install tmux
```

Run **sh swim.sh** in the **SCRIPTS** directory to start up the session manager running **tmux**.

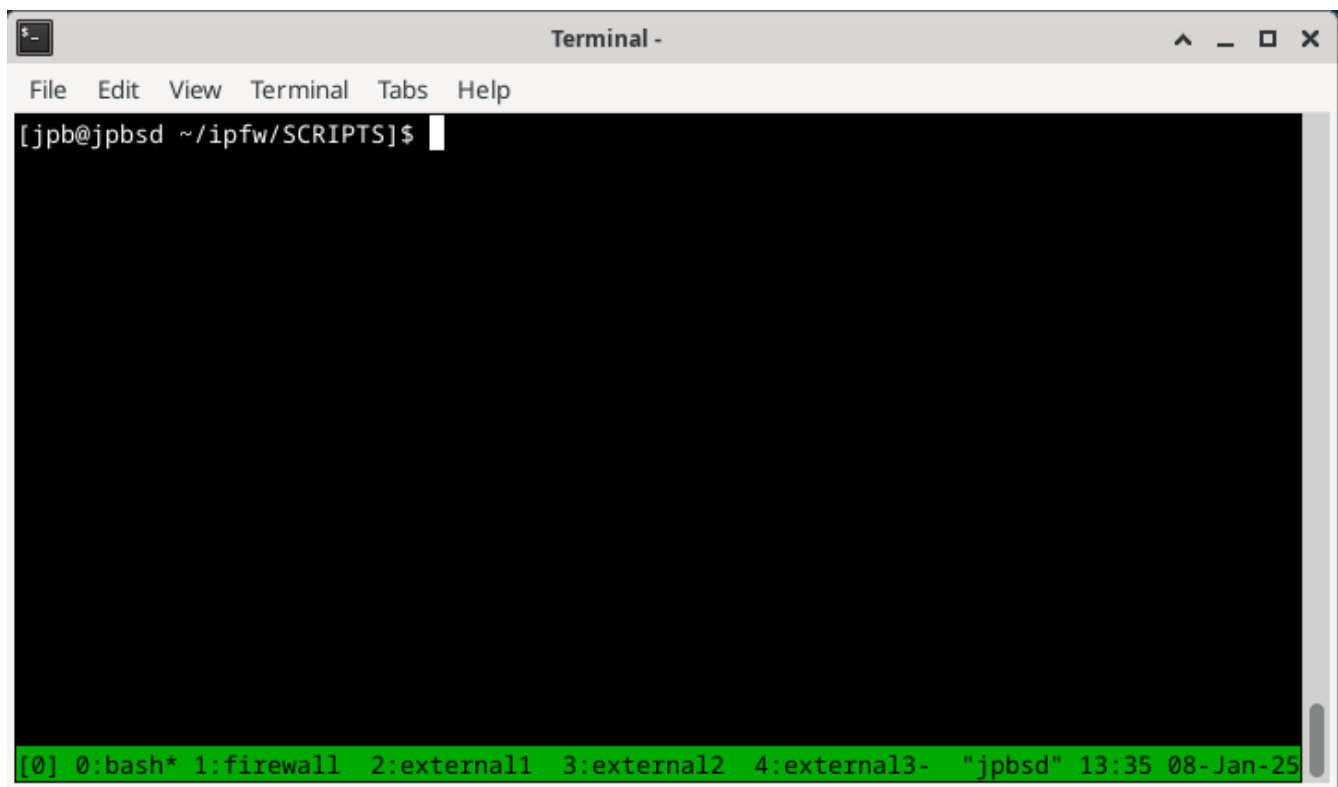


Figure 46. Starting Up **tmux(1)** Session Manager

The figure shows five named windows in one session (session [0]) with the **tmux** status line in green at the bottom:

Simplified **tmux(1)** Usage

tmux has a hierarchical organization.

1. **Sessions** - starts a session implicitly when **tmux** is run
 - a. **Windows** - adds a window with the command "tmux add-window" which can take a number of parameters **tmux** allows multiple windows in a session at the same time. The current window is starred ("*") in the status bar at the bottom of the main window.
 - i. **Panes** - splits a window into one or more panes either horizontally, or vertically

The **swim.sh** script shows how to set up one session with several windows. Panes are not used.

tmux is controlled by the user typing in any open window or pane. **tmux** uses `Ctrl+b` as its control key. Give **tmux** commands by typing the control key followed by one or more letters. To move from window to window use `Ctrl+b n` to move to the next window or `Ctrl+b p` to move to the previous window. Use `Ctrl+b ?` for a list of all key bindings.

Type **tmux kill-server** in any session window to completely leave **tmux**.

Consult the **tmux** manual page [tmux\(1\)](#) for more usage details.

Accessing the QEMU Serial Consoles

To access the VM serial consoles, move to the indicated window and telnet to the port on the local host for that VM:

```
Move to the external1 window in tmux, then
```

```
~/ipfw/SCRIPTS $ telnet localhost 4410
Trying ::1...
Connected to localhost.
Escape character is '^]'.
```

```
FreeBSD/amd64 (external1) (ttyu0)
```

```
login:
```

The **swim.sh** script has the following un-commented lines. Un-comment additional lines as needed

- **0:bash** - a terminal window of the user running **swim.sh**
- **1:firewall** - a terminal window to access the **firewall** VM
- **2:external1** - a terminal window to access the **external1** VM
- **3:external2** - a terminal window to access the **external2** VM
- **4:external3** - a terminal window to access the **external3** VM

The current window is marked with the '*' character in the status bar.

Note that the **swim.sh** script has entries for all windows used in this book. Uncomment the entries needed.

Run **sh swim.sh** in the **SCRIPTS** directory to start up the session manager running **tmux**.

D.2. Using [screen\(1\)](#) for Managing Serial Terminals

Install **screen** with:

```
# pkg install screen
```

screen, like **tmux**, is a terminal window manager. **screen** has its own control key - `Ctrl+a`, and like

tmux a list of key bindings is available at `Ctrl+a ?`.

By default, it does not use a status line, and once activated, it looks like no manager is active at all. Type **screen -ls** to determine if there is an active screen session running.

To display a status line in a live session use:

```
Ctrl+a : hardstatus alwayslastline
```

```
Ctrl+a : hardstatus string "%{= bw}%-w%{= rW}[%n %t]%-}%+w %=%{= kW} %H | %Y-%m-%d %c"
```

To always display the highlighted status line, edit the `.screenrc` file in the user `'$HOME'` directory and add:

```
hardstatus alwayslastline
hardstatus string "%{= bw}%-w%{= rW}[%n %t]%-}%+w %=%{= kW} %H | %Y-%m-%d %c"
```

To close all **screen** windows immediately and exit:

```
% screen -X quit
```

The **scim.sh** script has the following un-commented lines. Un-comment additional lines as needed

- **0:bash** - a terminal window of the user running **scim.sh**
- **1:firewall** - a terminal window to access the **firewall** VM
- **2:external1** - a terminal window to access the **external1** VM
- **3:external2** - a terminal window to access the **external2** VM
- **4:external3** - a terminal window to access the **external3** VM

The current window is highlighted in the status bar.

Note that the **scim.sh** script has entries for all windows used in this book. Uncomment the entries needed.

Run **sh scim.sh** in the `SCRIPTS` directory to start up the session manager running **screen**.

Appendix E: Appendix E: DNS Server Configuration

DNS configuration for IPFW Primer book.

Manifest of dnshost scripts and file.

File: dnshost_usrlocaletc_namedb.tgz

Description: Contains the configuration for the BIND 9 DNS services that run on this machine.

Installation:

Install bind9 package first, then untar this collection as follows:

```
# cd /usr/local/etc
# tar xvzf dnshost_usrlocaletc_namedb.tgz
```

Contents:

```
% tar tvzf dnshost_usrlocaletc_namedb.tgz
drwxr-xr-x  0 root  wheel      0 Nov 19 12:00 namedb/
-rw-r--r--  0 bind  bind    2403 Nov 19 11:59 namedb/bind.keys
drwxr-xr-x  0 bind  bind      0 Nov 19 11:59 namedb/dynamic/
-rw-r--r--  0 bind  bind    2618 Dec  2 12:34 namedb/named.conf
-rw-r--r--  0 bind  bind   21992 Nov 19 11:59 namedb/named.conf.sample
-rw-r--r--  0 bind  bind     927 Nov 19 11:59 namedb/named.root
-rw-r--r--  0 bind  bind    3317 Nov 19 11:59 namedb/named.root.SAVE
drwxr-xr-x  0 bind  bind      0 Dec  2 15:35 namedb/primary/
-rw-----  0 bind  bind     100 Nov 19 11:59 namedb/rndc.key
drwxr-xr-x  0 bind  bind      0 Nov 19 11:59 namedb/secondary/
drwxr-xr-x  0 bind  bind      0 Nov 19 11:59 namedb/working/
-rw-r--r--  0 bind  bind     297 Nov 19 11:59 namedb/working/managed-keys.bind
-rw-r--r--  0 bind  bind     355 Nov 19 11:59 namedb/primary/ptr_198.51
-rw-r--r--  0 bind  bind     465 Nov 19 11:59 namedb/primary/ptr_203.0
-rw-r--r--  0 bind  bind     693 Dec  1 19:29 namedb/primary/example.com
-rw-r--r--  0 bind  bind     148 Nov 19 11:59 namedb/primary/empty
-rw-r--r--  0 bind  bind     407 Nov 19 14:12 namedb/primary/ptr_ipv6
-rw-r--r--  0 bind  bind     287 Dec  2 15:35 namedb/primary/managed-keys.bind
-rw-r--r--  0 bind  bind     226 Nov 19 11:59 namedb/primary/localhost-reverse
-rw-r--r--  0 bind  bind     158 Nov 19 11:59 namedb/primary/localhost-forward
-rw-r--r--  0 bind  bind     351 Dec  1 19:30 namedb/primary/ptr_192.168
%
```

=====

bind.keys

```
# Copyright (C) Internet Systems Consortium, Inc. ("ISC")
#
# SPDX-License-Identifier: MPL-2.0
#
# This Source Code Form is subject to the terms of the Mozilla Public
# License, v. 2.0. If a copy of the MPL was not distributed with this
# file, you can obtain one at https://mozilla.org/MPL/2.0/.
#
# See the COPYRIGHT file distributed with this work for additional
# information regarding copyright ownership.

# The bind.keys file is used to override the built-in DNSSEC trust anchors
# which are included as part of BIND 9. The only trust anchors it contains
# are for the DNS root zone ("."). Trust anchors for any other zones MUST
# be configured elsewhere; if they are configured here, they will not be
# recognized or used by named.
#
# To use the built-in root key, set "dnssec-validation auto;" in the
# named.conf options, or else leave "dnssec-validation" unset. If
# "dnssec-validation" is set to "yes", then the keys in this file are
# ignored; keys will need to be explicitly configured in named.conf for
# validation to work. "auto" is the default setting, unless named is
# built with "configure --disable-auto-validation", in which case the
# default is "yes".
#
# This file is NOT expected to be user-configured.
#
# Servers being set up for the first time can use the contents of this file
# as initializing keys; thereafter, the keys in the managed key database
# will be trusted and maintained automatically.
#
# These keys are current as of Mar 2019. If any key fails to initialize
# correctly, it may have expired. In that event you should replace this
# file with a current version. The latest version of bind.keys can always
# be obtained from ISC at https://www.isc.org/bind-keys.
#
# See https://data.iana.org/root-anchors/root-anchors.xml for current trust
# anchor information for the root zone.

trust-anchors {
    # This key (20326) was published in the root zone in 2017.
    . initial-key 257 3 8
    "AwEAAaz/tAm8yTn4Mfeh5eyI96WSVexTBAvkMgJzkKTOiW1vkIbzxef3
    +/4RgW0q7HrxRixHlFlExOLAjr5emLvN7SWXgnLh4+B5xQlNVz80g8kv
    ArMtNROxVQuCaSnIDdD5LKyWbRd2n9WGe2R8PzgCmr3EgVlRjyBxWezF
    0jLHwVN8efS3rCj/EWgvIWgb9tarpVUDK/b58Da+sqqls3eNbuV7pr+e
    oZG+SrDK6nWeL3c6H5Apxz7LjVc1uTIdSIXxuOLYA4/ilBmSVIzuDWfd
    RUfhHdY6+cn8HFRm+2hM8AnXGXws9555KrUB5qihylGa8subX2Nn6UwN
    R1AKUTV74bU=";
```

```
};
```

```
=====
```

named.conf

```
// Refer to the named.conf(5) and named(8) man pages, and the documentation  
// in /usr/local/share/doc/bind for more details.
```

```
acl trusted-queriers {  
    203.0.113.0/24;  
    2001:db8:12::/64;  
    127.0.0.1;  
    ::1;  
    localhost;  
};
```

```
acl v6only-networks {  
    2001:db8:12::/64;  
};
```

```
options {  
    directory "/usr/local/etc/namedb/primary";  
    pid-file "/var/run/named/pid";  
    dump-file "/var/dump/named_dump.db";  
    statistics-file "/var/stats/named.stats";  
    listen-on { any; };  
    listen-on-v6 { any; };  
    recursion no;  
    allow-transfer { trusted-queriers; };
```

```
// NOTE: Remove comments when using DNS64  
// dns64 64:FF9B::/96 {  
// clients { any; };  
// exclude { 64:FF9B::/96; ::ffff:0000:0000/96; };  
// suffix ::;  
// };
```

```
};
```

```
// forward zone  
zone "example.com" {  
    type primary;  
    file "/usr/local/etc/namedb/primary/example.com";
```

```

    allow-transfer {trusted-queriers;};
};

// reverse zones for 203.0, 198.51, 192.168, and 2001:0db8
zone "0.203.in-addr.arpa" {
    type primary;
    file "/usr/local/etc/namedb/primary/ptr_203.0";
    allow-transfer {trusted-queriers;};
};

zone "51.198.in-addr.arpa"{
    type primary;
    file "/usr/local/etc/namedb/primary/ptr_198.51";
    allow-transfer {trusted-queriers;};
};

zone "2.1.0.0.8.b.d.0.1.0.0.2.ip6.arpa" {
    type primary;
    file "/usr/local/etc/namedb/primary/ptr_ipv6";
    allow-transfer {trusted-queriers; };
};

zone "168.192.in-addr.arpa" {
    type primary;
    file "/usr/local/etc/namedb/primary/ptr_192.168";
    allow-transfer {trusted-queriers;};
};

// Block below added by BIND9
// RFCs 1912, 5735 and 6303 (and BCP 32 for localhost)
zone "localhost"      { type primary; file "/usr/local/etc/namedb/primary/localhost-
forward"; };
zone "127.in-addr.arpa" { type primary; file "/usr/local/etc/namedb/primary/localhost-
reverse"; };
zone "255.in-addr.arpa" { type primary; file "/usr/local/etc/namedb/primary/empty"; };
// RFC 1912-style zone for IPv6 localhost address (RFC 6303)
zone "0.ip6.arpa"    { type primary; file "/usr/local/etc/namedb/primary/localhost-
reverse"; };
// "This" Network (RFCs 1912, 5735 and 6303)
zone "0.in-addr.arpa"  { type primary; file "/usr/local/etc/namedb/primary/empty"; };

// Our own root zone file so we don't leak out to the Internet
zone "." {
    type master;
    file "/usr/local/etc/namedb/named.root";
    allow-transfer {trusted-queriers; };
};

```



```
=====
```

named.root

```
; This file holds the information on root name servers needed to
; initialize cache of Internet domain name servers
; (e.g. reference this file in the "cache . <file>"
; configuration file of BIND domain name servers).
;
; This file is made available by InterNIC
; under anonymous FTP as
;   file           /domain/named.cache
;   on server      FTP.INTERNIC.NET
;   -OR-          RS.INTERNIC.NET
;
; last update:    November 16, 2017
; related version of root zone:  2017111601
;
; FORMERLY NS.INTERNIC.NET
;
```

```
$TTL 3600
```

```
. 3600 IN SOA dnshost.example.com. jpb.dnshost.example.com (
    100      ; serial
    14400   ; refresh
    7200    ; retry
    28800   ; expire
    64000   ) ; min neg cache expire

. 3600 NS dnshost.example.com.
dnshost.example.com. 3600 A 203.0.113.53
dnshost.example.com. 3600 AAAA 2001:db8:12::53
```

```
=====
```

rndc.key

```
key "rndc-key" {
    algorithm hmac-sha256;
    secret "wesiGsTgu70wV44aA6C2P8XmZdW4z/YdPJ4D/vRNPTM=";
};
```

```
=====

empty
```

```
$TTL 3h
@ SOA @ nobody.localhost. 42 1d 12h 1w 3h
    ; Serial, Refresh, Retry, Expire, Neg. cache TTL

@ NS @

; Silence a BIND warning
@ A 127.0.0.1
```

```
=====

example.com
```

```
$TTL 3600
@ IN SOA example.com. jpb.example.com. (
    5 ; Serial
    3h ; Refresh
    1h ; Retry
    1w ; Expire
    1h ) ; Negative Cache TTL
;
; name servers - NS records
@ IN NS dnshost.example.com.

; name servers - A records
dnshost IN A 203.0.113.53
;external1 IN A 203.0.113.10
external1 IN A 192.168.1.2
external2 IN A 203.0.113.20
external3 IN A 203.0.113.30
firewall IN A 203.0.113.50
firewall-em0 IN A 203.0.113.50
firewall-em1 IN A 198.51.100.50
firewall-em1 IN AAAA 2001:db8:12::50
internal IN A 198.51.100.200

; name servers - AAAA records
```



```
0 ; retry (0 seconds)
0 ; expire (0 seconds)
0 ; minimum (0 seconds)
)
KEYDATA 20241202213508 19700101000000 19700101000000 0 0 0 ; placeholder
```

ptr_192.168

```
$TTL 3600
@ IN SOA example.com. jpb.example.com. (
    3 ; Serial
    3h ; Refresh
    1h ; Retry
    1w ; Expire
    1h ) ; Negative Cache TTL
```

;

```
; name servers - NS records
    IN NS dnshost.example.com.
```

```
; PTR Records
```

```
53.1 IN PTR dnshost.example.com.
2.1 IN PTR external1.example.com.
```

ptr_198.51

```
$TTL 3600
@ IN SOA example.com. jpb.example.com. (
    3 ; Serial
    3h ; Refresh
    1h ; Retry
    1w ; Expire
    1h ) ; Negative Cache TTL
```

;

```
; name servers - NS records
    IN      NS      dnshost.example.com.
```

```
; PTR Records
```

```
50.100 IN  PTR firewall-em1.example.com.
200.100 IN  PTR internal.example.com.
```

```
=====
```

ptr_203.0

```
$TTL    3600
@       IN      SOA    example.com. jpb.example.com. (
        3      ; Serial
        3h    ; Refresh
        1h    ; Retry
        1w    ; Expire
        1h )   ; Negative Cache TTL
;
```

```
; name servers - NS records
    IN      NS      dnshost.example.com.
```

```
; PTR Records
```

```
53.113 IN  PTR dnshost.example.com.
10.113 IN  PTR external1.example.com.
20.113 IN  PTR external2.example.com.
30.113 IN  PTR external3.example.com.
50.113 IN  PTR firewall-em0.example.com.
```

```
=====
```

ptr_ipv6

```
$TTL    3600
@       IN      SOA    example.com. jpb.example.com. (
```

```

        3      ; Serial
        3h    ; Refresh
        1h    ; Retry
        1w    ; Expire
        1h )  ; Negative Cache TTL
;

@ IN NS dnshost.example.com.

$ORIGIN 0.0.0.0.2.1.0.0.8.b.d.0.1.0.0.2.ip6.arpa.
3.5.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0 IN PTR dnshost.example.com.
6.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0 IN PTR v6only.example.com.

```

managed-keys.bind

```

$ORIGIN .
$TTL 0 ; 0 seconds
@ IN SOA . . (
        100      ; serial
        0        ; refresh (0 seconds)
        0        ; retry (0 seconds)
        0        ; expire (0 seconds)
        0        ; minimum (0 seconds)
    )
    KEYDATA 20220502020339 19700101000000 19700101000000 0 0 0 ; placeholder

```

Index

@

- blockdev, 168
- device, 168
- netdev, 168
- .screenrc, 227
- /boot/loader.conf, 171
- /etc/syslog.conf, 170
- 464XLAT, 107, 136, 140
- 5-tuple, 95
- 64:ff9b::, 127, 130, 132, 134, 134

A

- abort, 144
- abort6, 144
- add, 17
- add a serial console, 171
- address
 - list, 29
 - range, 29
 - range increasing, 30
 - sparse addressing, 29
- alias, 157
- aliasing, 122
- allow/accept/pass/permit, 18
- allow_private, 131, 134
- antispoof, 153
- Appendix A, 107

B

- bandwidth, 87, 95, 95
- bind9, 5, 125, 129
- binding
 - key, 172, 226
- BMP, 107
- bridge, 8, 78, 111, 148, 162, 168, 170, 176
- bsdclat464.sh, 138, 169
- bsdplat464.sh, 169
- buckets, 93, 94
- burst, 93

C

- call/return, 18, 60
 - endless loop, 61
 - error, 62
 - example, 60, 61

- general notes, 63
- syntax, 61

- check-state, 18
- CLAT, 135, 136
- clear counters, 26
- cmake, 169
- cmdwatch, 59, 168
- comconsole, 171
- console
 - serial, 107, 170, 171, 171, 173, 226
 - virtual, 170, 170
- console.log, 170
- control key, 172, 226, 227
- count, 18
- counters, 26

D

- delay, 93, 96
- delete, 17
- deny/drop, 18
- deny_in, 123
- developers, 5
- DHCP, 168, 170, 176
- DISPLAY, 166
- divert, 18, 47, 53
 - example, 54
 - execution, 55
 - general notes, 57
 - object, 55
 - permission denied, 56
 - ruleset, 54
 - socket, 53
 - socket notes, 58
 - source code, 53
 - view with netstat, 54
- divert.c, 178
- dnctl, 88
- DNS64, 106, 125, 126, 129, 130, 131
- dnshost.sh, 178
- doas, 8
- dst-port, 31
- dumb terminal, 171
- dummynet, 84, 85, 87, 89, 95
- dynamic pipe, 101
- dynamic queue, 95, 95

E

ECN, [49](#)
enable/disable, [17](#)
external1.sh, [178](#)
external2.sh, [178](#)
external3.sh, [178](#)

F

fbsd.iso, [176](#)
fetch, [167](#)
FIFO, [91](#), [94](#)
firewall.sh, [178](#)
firewall2.sh, [178](#)
flags
 SYN, [112](#), [113](#), [126](#)
flow mask, [95](#), [96](#), [100](#)
flow queue, [94](#)
flow rate, [103](#)
flush, [17](#)
full screen, [170](#)

G

git, [169](#)
github, [9](#), [177](#)
global, [124](#)
goodput, [85](#), [87](#)
GUI, [8](#)

H

hping3, [11](#), [149](#), [150](#), [151](#), [168](#)

I

IANA, [26](#)
ICMP, [47](#), [48](#)
 reply, [48](#)
icmptypes, [132](#)
internal.sh, [178](#)
iperf3, [11](#), [85](#), [85](#), [88](#), [89](#), [95](#), [97](#), [98](#), [98](#), [98](#), [103](#),
 [168](#)
ipfw
 -D, [21](#)
 -d, [21](#), [63](#)
 -S, [34](#)
 -SaD, [59](#)
 divert, [53](#)
 show, [51](#)
ipfw0, [40](#)
ipfw_nat, [115](#)

ipfw_nat64stl, [134](#)
ipid, [149](#)
iplen, [149](#)
ipprecedence, [149](#)
ipttl, [149](#), [151](#)
ipv6, [5](#), [125](#)
ISO, [107](#), [167](#)

J

jail, [157](#)
 configuration, [159](#), [162](#)
 jailid, [161](#)
 name, [161](#)
 VNET, [161](#)
 vnet, [162](#), [163](#), [164](#)
jail1.sh, [178](#)
jot, [86](#)
juniper, [135](#)

K

keyword
 and, [28](#)
 any, [28](#)
 call, [60](#)
 divert, [47](#), [53](#)
 flush, [38](#)
 gid, [63](#)
 limit, [58](#)
 log, [41](#)
 logamount, [44](#)
 lookup, [65](#)
 me, [28](#)
 not, [28](#)
 ports, [30](#)
 prob, [32](#)
 redirect_addr, [121](#)
 reset, [46](#)
 resetlog, [44](#)
 return, [60](#)
 sctp, [75](#)
 set, [33](#)
 setdscp, [49](#)
 skipto, [50](#)
 table, [66](#), [66](#)
 tablearg, [68](#)
 tee, [47](#)
 uid, [63](#)
 unreach, [47](#)

valtype, 71

L

libalias, 108

limit, 18, 58

example, 59

flow element, 58

sample run, 59

list, 17

load balancing, 122

load sharing NAT, 106

log, 26

.debug level, 42

.info level, 42

/etc/syslog.conf, 42

count actions, 40

ipfw0, 41

LOG_SECURITY, 42

logamount, 44, 45

match, 43

Method 1, 41

Method 2, 42

mutually exclusive, 44

reading with tail -f, 43

reset, 47

resetlog, 44, 44

syslog buffering, 43

syslogd, 43

log file, 145

log/logamount, 18

logging, 40

example, 40

ipfw0, 40

syslog, 40

lookup, 18

key, 65

value, 65

lookup table, 29, 65

addr, 65

create, 66

flow, 66, 67, 71, 72

general notes, 73

iface, 66

MAC, 66

number, 66

type notes, 74

types, 65

uses, 67

lookup tables, 18

LSNAT, 106, 116, 116, 119, 121

lynx, 119, 120, 121, 132, 134, 143, 168

M

mark, 144, 146

mask, 95

me6, 28

mkbr.sh, 79, 108, 109, 117, 145, 145, 148, 178

module

dummynet, 95

ipdivert, 54, 58

ipfw, 8, 12, 110, 131, 134, 148

ipfw_nat, 8, 110, 119

ipfw_nat64, 131, 134

ipfw_nptv6, 8, 148

sctp, 75, 78

mouse

grab, 170

N

NAT, 106, 108, 110

object, 110

static, 111

NAT64, 106, 125, 125, 131, 131, 133, 135

nat64clat, 143

NAT64LSN, 125

nat64lsn, 131, 143

NAT64STL, 133

natd, 108, 115

nc, 164

ncat, 8, 20, 31, 39, 49, 59, 149, 150, 151, 153, 156,
157

netcat, 8

networks

private, 170

nginx, 5, 118, 139, 168

nmap, 8, 168

NPTv6, 146, 149

O

omit flag, 97

operator

or, 27

or-block, 27, 28

P

pane, 225

ping, [113](#), [115](#), [118](#), [122](#), [140](#)
ping6, [132](#), [134](#)
pipe, [84](#), [87](#), [89](#), [90](#), [91](#), [92](#), [100](#), [102](#), [103](#), [110](#)
 delete, [105](#), [105](#)

PLAT, [136](#)

port number, [98](#)

ports

 /etc/services, [30](#)

 examples, [30](#)

 ranges, [30](#)

privilege, [8](#)

prob, [18](#), [32](#), [122](#)

protocol

 /etc/protocols, [26](#)

 keywords, [27](#)

 list, [27](#)

 other examples, [58](#)

proxy_only, [124](#)

Q

qcow2, [168](#), [176](#)

QEMU, [8](#), [9](#), [166](#), [167](#), [171](#)

 pkg install, [167](#)

qemu-img, [168](#)

qfq, [93](#)

QFQ algorithm, [92](#)

queue, [84](#), [87](#), [91](#), [93](#), [94](#), [95](#), [97](#), [100](#), [100](#), [110](#)

 delete, [105](#)

queue mask, [98](#)

queue weight, [99](#)

Quick Start, [9](#), [16](#), [170](#), [173](#)

R

red, [93](#), [100](#)

redirect_address, [116](#)

redirect_port, [116](#)

redirect_proto, [116](#)

reset, [18](#), [18](#), [46](#), [123](#)

 view, [46](#)

reverse, [123](#)

RFC

 1918, [131](#)

 2391, [116](#)

 2474, [49](#)

 3260, [49](#)

 3849, [10](#), [125](#), [129](#)

 4960, [75](#)

 5737, [10](#), [129](#)

6052, [125](#)

6146, [125](#), [126](#)

6147, [127](#)

6296, [146](#), [149](#)

6877, [135](#), [135](#)

6890, [131](#)

791, [150](#)

Round Robin scheduler, [92](#)

rule

 add, [19](#)

 addresses, [28](#)

 allow, [19](#)

 basic keywords, [17](#)

 comments, [33](#)

 delete, [23](#)

 dynamic, [19](#), [20](#)

 example syntax, [16](#)

 flush, [24](#)

 general format, [16](#)

 keep-state, [19](#)

 list, [24](#)

 log, [40](#)

 manually create dynamic, [21](#)

 number list, [24](#)

 number range, [24](#)

 numbering, [21](#)

 rule body, [16](#)

 same number, [22](#)

 setup, [19](#)

 show, [24](#), [25](#)

 syntax constructs, [19](#)

 tag, [39](#)

 viewing dynamic, [21](#)

 zero, [26](#)

 zero default rule, [26](#)

rules, [8](#)

ruleset, [8](#), [11](#), [90](#)

 development, [17](#)

runvm.sh, [79](#), [108](#), [109](#), [117](#), [148](#), [178](#)

S

same_ports, [123](#)

sched, [84](#), [87](#), [91](#), [92](#)

 delete, [105](#)

scheduler type, [105](#)

scheduling algorithm, [85](#)

scim.sh, [172](#), [178](#), [227](#), [227](#), [227](#)

screen, [171](#), [171](#), [226](#), [227](#)

SCRIPTS, 107, 167

scripts, 8

SCTP, 75

- 4-way handshake, 76
- association, 76
- building usrsctp, 80
- tsctp example, 79
- tsctp test tool, 77
- two-rule version, 80
- typical packet, 77
- UDP encapsulation, 81
- usrsctp, 80
- versions, 75
- view with netstat, 76

SDL, 166, 167

serial devices, 171

serial keyword, 171

session, 172, 172, 225, 227

set, 18

- disable, 34
- enable, 34
- set 0 default, 37
- set 31, 36, 37
- swap, 35
- swap disabled, 35

setdcsp

- code points, 49

setdscp, 18, 49

- ECN, 49
- service classes, 49

setmark, 144, 146

sets, 33

- caveats, 33

setup, 120

show, 17

simple NAT, 106

skip_global, 124

skipto, 18, 50

- general notes, 51

slow start, 97

Special Use Addresses, 10

src-port, 31

stateful translation, 125

stateless translation, 125

static queue, 95

status bar, 172, 226, 227

steady state, 97

sudo, 8, 167

swim.sh, 172, 178, 226

sysctl

- net.inet.ip.forwarding, 110, 138, 139
- net.inet.ip.fw.autoinc_step, 21
- net.inet.ip.fw.dyn_count, 59
- net.inet.ip.fw.dyn_udp_lifetime, 59
- net.inet.ip.fw.nat64_debug, 132, 134
- net.inet.ip.fw.nat64_direct_output, 132, 134, 138
- net.inet.ip.fw.one_pass, 148
- net.inet.ip.fw.tables_max, 74
- net.inet.ip.fw.tables_sets, 74
- net.inet.ip.fw.verbose, 40, 44, 132, 134, 145, 148, 151
- net.inet.ip.fw.verbose_limit, 45
- net.inet.ip6.forwarding, 138, 139, 148

syslog, 145, 170

- /etc/syslog.conf, 40

syslogd, 149, 170

T

T1 line, 90

table, 29

- destroy, 68
- example, 70, 71
- flow, 72
- flow examples, 72
- flow table, 71
- limit example, 71
- value, 67
- with skipto, 69

tablearg, 68, 124

- example, 68

tag, 18, 39

- example, 39

tap, 8, 78, 148, 168, 170, 176

tcon.sh, 11, 13, 19, 34, 50, 178

tconr.sh, 11, 178

tcont.sh, 11, 178

TCP

- 3-way handshake, 20, 63

tcpack, 152

tcpdatalen, 152

tcpdump, 111, 112, 112, 114, 115, 122, 142, 149, 151

tcpflags, 152

tcpmss, 152

tcpoptions, 152

tcpseq, 152
tcpwin, 152
tee, 18, 47
telnet, 111, 112, 113, 171, 171, 173
throughput, 86
tmux, 171, 171, 171, 172, 173, 225, 225, 225, 226
traffic shaping, 87
transfer speed, 86
tserv.sh, 11, 12, 19, 19, 152, 160, 178
tserv3.sh, 11, 50, 178
ttl, 150, 151

U

ucon.sh, 11, 70, 178
uconr.sh, 11, 25, 178
ucont.sh, 11, 32, 41, 145, 178
UFS, 168, 176
uid/gid, 63

- example, 64
- general notes, 65
- ICMP issue, 65
- usage, 64

unreach, 18, 47
unreg_cgn, 114
unreg_only, 114
userv.sh, 11, 32, 64, 145, 156, 157, 178
userv3.sh, 11, 24, 41, 55, 59, 61, 61, 70, 178
userv5.sh, 178

V

v6only.sh, 178
verrevpath, 153
versrcreach, 153
virtual machine, 8, 166
virtual terminal, 166
virtualization, 166
VM, 107
vm_envs.sh, 178
VM_SCRIPTS, 169

W

weight, 93, 95, 96, 100, 100
Well Known Prefix, 127, 131
WF2Q+ algorithm, 92, 92
wfq2, 105
window, 225

X

XLAT464, 135
XLAT464 CLAT, 135

Z

ZFS, 168, 176