A Crash Course in Linux Networking

David Guyton

April 2018 (edited August 2019)

Table of Contents

Preamble 2

- My Ubuntu Obsession 2

Routing and Filtering Network Traffic 3

A Brief History of Linux Networking 5

- ipchains 5
- The RPDB Revolution 6
- netfilter 6
- iptables 6

Network Routes 9

How Linux Routes Network Packets 11

- Incoming Packets 11
- Forwarded Packets 11
- Outgoing Packets 11
- The Whole Enchilada in One Picture 11

Policy Based Routing 14

- The RPDB Triad: Rules, Routes, and Tables 15
- Routes 16
- Route Priority Processing 23
- IP Route Command Examples 23
- Source NAT in ip route 25
- Deleting Routes 26
- Creating New Routing Tables 26
- Special Routing Use-Case Scenarios 28
- The Route Command: An Oldie But a Goodie 31
- Netstat 33
- Flags: Hosts, Gateways, and Routes 35
- Split Gateways 35

Follow the Rules 40

- IP Rule Syntax 40
- IP Rule Examples 42

Rule Priority 43
Routing Marked Packets with fwmark 45
iptables Explained 46
Terminology Confusion 46
Chains and Tables 46
Chains 47
Tables 50
Viewing Existing iptables Rules 53
How to Delete Chains 55
iptables Process Flow: Chains, Tables, and Rules 57
CONNTRACK 58
Custom Chains 58
Chain Policies 59
iptables Command Line Syntax 60
Applying iptables Commands to Chains 61
Parameters 61
iptables Extensions 62
Actions and Targets 63
Match Extensions 64
Target Extensions 68
Extensions and Chains: Where and When to Apply *iptables* Rules 73
Inverse Operand 74
Noteworthy Variable Syntax 74
Connection Tracking 74
Protocols 75
Testing Your Rules 76
Listing Rules by Number 76
Create Your Shell Scripts 76
Helpful Tips 79
Don't Forget to Flush! 79
Do NOT Flush ip rules 79
mark vs. fwmark: What's the Difference? 79

Persisting Routes, Rules, and Tables Across Reboots 79
SystemD and the Start-up Process 79
References 81
Endnotes 85

Table of Figures

Figure 1: NetFilter Topology ________ *4*
Figure 2: Historical Timeline of iptables ________ *5*
Figure 3: Top Level NetFilter Hooks: NFTables, UFW ________ *7*
Figure 4: Linux Networking Concepts and Associated Tools ________ *9*
Figure 5: Packet Routing Tree ________ *10*
Figure 6: iptables Input and Forward Chains Process Flow Diagram ________ *12*
Figure 7: iptables Output chain process flow ________ *13*
Figure 8: AND Bit Logic Illustration ________ *37*

A Crash Course in Linux Networking

The primary purpose of this guide is to familiarize you with some of the most important concepts in Linux networking. I'm not endeavoring to teach you everything, nor provide you *all* the details of the tools described herein. My goal is to provide enough information to get you knee deep into some subjects, and dip your toe in the water with others. Most people don't require a true deep-dive. This guide is more broad than deep. It covers common topics and provides the reader with more in depth detail than most other authors reveal.

I hope you find it beneficial, and appreciate any constructive feedback. You may reach me via my blog, https://datahacker.blog

- David Guyton

Preamble

There are plenty of guides online if you want to learn about networking in Linux. I wrote this particular manuscript while working on a “How-To” instructional guide on Linux media server construction and configuration. Borne out of my research of client-side VPN configurations, I hope it spares you the many hours I spent developing a basic understanding of Linux networking, including *iptables*, *iproute*, and *netfilter* just to name a few components.

This isn't meant to be an all encompassing guide to Linux networking or any component of it. Rather, it is a mini-guide designed to help you gain a basic technical understanding of the topics contained herein. Even if you're not building or maintaining a firewall, other features such as VPN configuration require a rudimentary understanding of the underlying concepts of both VPNs and networking.

In this document I'll cover:

1. Routing network traffic
2. Packet manipulation
3. How Linux structures network management
4. Network management tools
5. Traffic filtering vs. routing

And before diving in, let's be clear on what I am **not** covering! First, I am not reviewing ALL features, functions, options, and parameters of Linux networking. That is an incredibly huge topic. You can't even fit it all in a single book. A series of tomes would be more like it, so if you need deep levels of detail, please find the appropriate Wiki or other documentation. **This document is meant to be a primer on how networking is managed by Linux, and a reference for the most common associated tools and commands**.

My Ubuntu Obsession

In the Linux world, I'm most familiar with Ubuntu, which is why you'll find all the code examples contained herein based on that platform. While Ubuntu has its own personality and quirks, it is largely true to the Linux core, and it is still a Debian flavor. With a stable and well funded corporate sponsor and strong community, IMHO, it provides a great platform for tutorials such as this one.

As of this writing, Ubuntu 16.04.x LTS is my go-to version. I prefer to hang a back a bit from cutting-edge versions of any operating system. I value stability and the fact a modicum of bug-cleaning has already occurred over the “latest and greatest” version of almost anything. I submit to you that reliability does (or should) trump feature sets for almost all applications. Most operating systems need a minimum of a year in production before the most egregious bugs have been identified and squashed.

Routing and Filtering Network Traffic

Network packet routing in Linux is - unfortunately - an area I believe generates considerable confusion when discussing routing and routing policies. At a high level, Linux network routing is divided into two core sections. The first is the actual routing policy process and the other is a packet mangling process. Together they provide a very powerful combination of tools that far exceeds the capabilities of traditional table-based network routing systems. When the Linux kernel has an outgoing data packet destined for a network device, how does it determine where to send the packet? When the Linux kernel receives a data packet from another device, how does it determine what to do with it?

To create a “smart” network routing system, you will need to grasp a few basic concepts of how Linux disseminates and processes network traffic. It's difficult to know where to begin. Linux networking is very complex, which is why there are so many discussions of the subject. I'll layout a top-down approach and basic framework for you. Hopefully, this will clarify things and make the process easier to understand. To manipulate your network traffic at will – such as choosing which traffic passes through a VPN and which does not – you need to understand how the Linux kernel routes network traffic, and the tools available to manipulate network routes.

First, by default, all network traffic is permitted. One can imagine how that is not an ideal approach under many circumstances. Developing a comprehensive plan for managing your network traffic - even for a stand-alone server - is not straightforward in Linux. You have two primary paths available to you: filtering and routing. You may choose to use one or the other, or both. Let's be clear on what 'filtering' and 'routing' mean when it comes to networking. "Filtering" is the manipulation or modifying of network packets coming into or leaving your server. A firewall is the best example I can think of to represent a filter. That's all a firewall does is filtering. "Routing" refers to the pathways packets may take when leaving your server. Network traffic is sent out on "routes." A route is a conduit to one or more other network devices.

The core of Ubuntu's filtering mechanism is *netfilter* - Linux' firewall. Ubuntu uses two common Linux application layers on top of *netfilter* called **iptables** and **ip route**. *Netfilter* itself has two components: an API (Application Programming Interface) with hooks to the kernel, and underlying network architecture code.

A completely independent tool in Ubuntu called **ufw (**"Uncomplicated FireWall") may also be used to manipulate *netfilter*. **Ufw** is a front-end for *iptables*. Some people find **ufw** easier to work with than *iptables*, but I'll be teaching you how to use the latter. Why? Yes, it's more complicated, but the upside is you'll get a better idea of what you are doing when manipulating packet data.

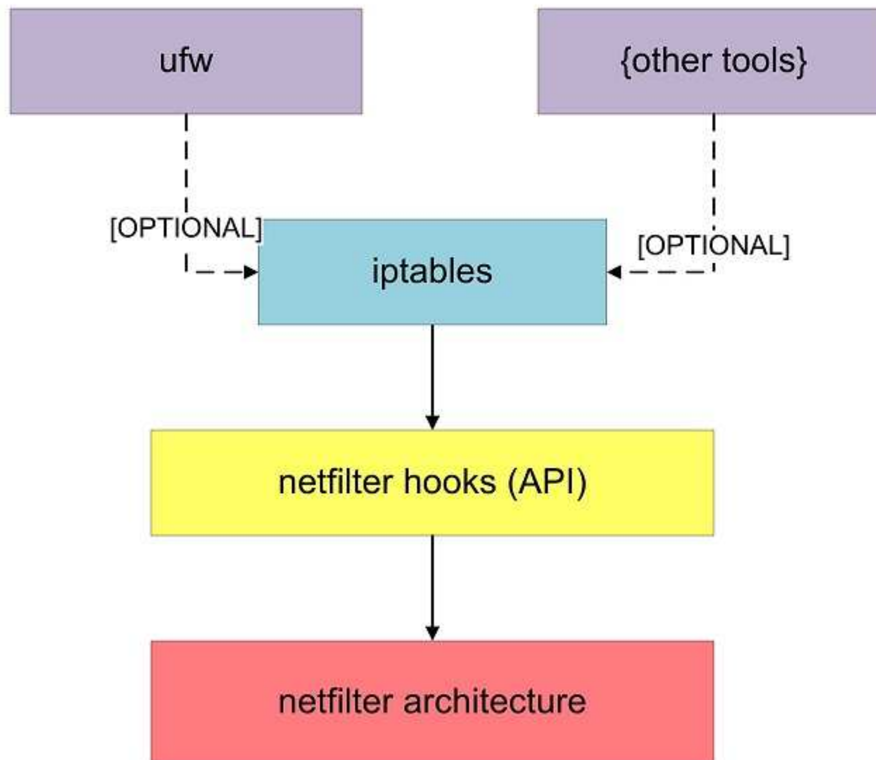

Figure 1: NetFilter Topology

So, what's the net benefit for you?

There are two portions to packet networking in Linux: *routing* and *filtering*. The routing process is a map of your network. Routes determine possible paths where a packet may be sent. It is the where in networking. You may think of it as a postal system, where a device's address on the network is similar to the address of a building in the postal service delivery system. Routing has to do with where things are going and where they came from.

Filtering governs what and how data is sent via a route. In the postal system analogy, filtering is concerned with what is being transported; the *packet*. *Netfilter* handles most of the packet mangling. Its scope is limited to packet manipulation. It doesn't cause a packet to move in or out of the server, and doesn't understand network topography like the routing process. On-the-other-hand, the routing process can only choose which path a packet takes based on a limited amount of information.

A Brief History of Linux Networking

Have you ever wondered why Linux network management is so confusing? You need only make a cursory review of the history of Linux networking tools to understand how it got this way.

Ironically, it seems to me that Linux network evolution is largely responsible for the confusion people often encounter when searching online for information on how Linux processes network routing filters and commands. This isn't surprising if you look at the history of significant routing behavior changes that occurred in the Linux kernel relatively close together from the mid-1990's to 2001. The code remained more or less untouched until 2014, and even then the milestone was an overhauled interface to the kernel. The core remained virtually unchanged. Even now – in 2018 – we are still using nearly 20-year old networking code in our Linux servers. Amazing when one takes into consideration the evolution in anecdotal technologies since then, such as network security, identity, and block chains to name a few.

Here is a synopsis of the historical timeline of Linux user-based networking tools:

Tool	Released	Kernel	Deprecated	Evolution
ipfwadm	1995	1.2.1	X	First packet filtering tool for Linux
ipfwadm (v2)	1996	1.3.66	X	Chains concept introduced (IN, OUT, FORWARD)
iproute	1997	2.0	X	Original ip utility (RPDB introduced)
ipchains	May 1999	2.2	X	User-created ipchains, fwmark added
iproute (v2)	Sep 1999			Revised ip utilities
iptables	Mar 2000	2.3		iptables released (replaces ipchains, fwmark)
	2001	2.4		iptables packet filtering and header mangling
nftables	Jan 2014	3.13		Replacement to iptables

Figure 2: Historical Timeline of iptables

Linux kernel 1.2.1 - released in 1996 - unveiled a firewall administration tool for the first time called **ipfwadm**, which was an abbreviation for IP FireWall ADMinistration. It was the first packet-level screening interface for Linux where a sysadmin could tweak how the kernel handled network packets. It was replaced a few years later with *ipchains*, which was itself subsequently replaced by *iptables* the following year.

Nftables is a replacement for *iptables*, along with other little-known networking tools such as *arptables* and *ebtables*. It combines the functionality of those tools into a single interface, which then hooks into *netfilter*. The main benefit to **nft** is the consolidation of the aforementioned tools plus extensions of these tools (e.g. CONNTRACK) into a single application.

ipchains

The *chains* concept in Linux networking is very primitive. It dates back to Linux kernel 1.3.66 in 1996 when the chains consisted of just three: IN, OUT, and FORWARD. The concept entails each of the three main processes in routing network packets through a server. Those packets will either be incoming to a local destination, outgoing to another device and originating from a local service, or outgoing to another device and not originating from a local service.

You may think of the chains as highways inside your server. They are able to direct traffic flow along a particular direction, but cannot modify the packets. Although primitive compared to more recent routing processes, *ipchains* remains engrained in the Linux' networking DNA.

The RPDB Revolution

Released in 1997, Linux kernel 2.0 was a watershed event for the Linux community. It included a complete rewrite of the networking backend. A new network management tool debuted. Called *iproute*, the Routing Policy DataBase (RPDB) was born.

The RPDB created the concept of a table of routing tables; a "master" routing table, if you will. Prior to the 2.0 kernel, Linux used the traditional model of a single routing table. Shortly afterward the advent of *ipchains* arrived with the 2.2 kernel, and in September 1999, version 2 of *iproute* was released. A set of tools originally written by Alexey Kuznetsov that expanded the capability of the RPDB, *iproute2* added an additional layer of rules which today we think of as filtering rules. This controversial change allowed some functions of *ipchains* to be duplicated by the RPDB. This in turn led to a split in philosophies of where and how network packet mangling and Network Address Translation (NAT) should be handled, and which tools ought to be interacting with *netfilter* (Linux' network management kernel layer). Still relevant today, when using **ip route** and **ip rule**, you're using functions of *iproute2*. It superseded several original *iproute* tools including **route**, **ifconfig**, and **netstat**.

Approximately six months later, another initiative called *iptables* went live with the 2.3 kernel. It replaced *ipchains* with a foundation for a more robust solution and more or less wrestled the reins of various functions away from *iproute* again. When the Linux 2.4 and 2.6 kernels were released a short time later, a number of functions of *iproute2* were deprecated, solidifying *iptables* lead in packet mangling and NAT functions. While *ipchains*' legacy of chains remained, *iptables* expanded the *chains* concept to five (5) built-in chains and created the ability of users to add new, custom chains to the network management process.

netfilter

Another point-of-confusion is *netfilter*. Perhaps because though it is spoken of regularly, yet there is no **netfilter** command. *Netfilter* is a process responsible for network packet management and NAT (Network Address Translation). It is effectively the network traffic cop inside your server.

netfilter is comprised of two parts: 1) a set of kernel hooks; and 2) a tool that translates commands to kernel actions. The various *xx*-**tables** tools interact with the translation portion of *netfilter*.

iptables

The next significant evolution in network packet management within Linux was iptables. An evolution of *ipchains* and still widely used today, *iptables* cannot direct traffic flow, but can modify network packets.

Tables are called by *chains* and provide a finer level of detailed packet filtering. The benefit of this approach is all filtering rules are located in one place. Packet directional flow control is handled by the *chains* and they in turn call the *tables* as needed. A *table* contains a set of packet filtering instructions, but cannot alter the direction the packet is flowing.

It is important to note although the command **ipchains** is deprecated in Linux, *chains* themselves still exist and are a critical component of *iptables*.

NFTables

Nftables is a replacement to *iptables*, *ip6tables,*[1] *arptables*,[2] and *ebtables.*[3]

Nftables functions differently from *iptables* and its extensions. It completely bypassing *netfilter*'s translation tool and passing data directly to *netfilter*'s kernel interface. To put this in context, if you were to remove *netfilter*, *nftables* would continue to function properly but *iptables* and its *xx*-**tables** associated commands would break.

nftables bypasses the traditional *iptables* hooks into *netfilter*.

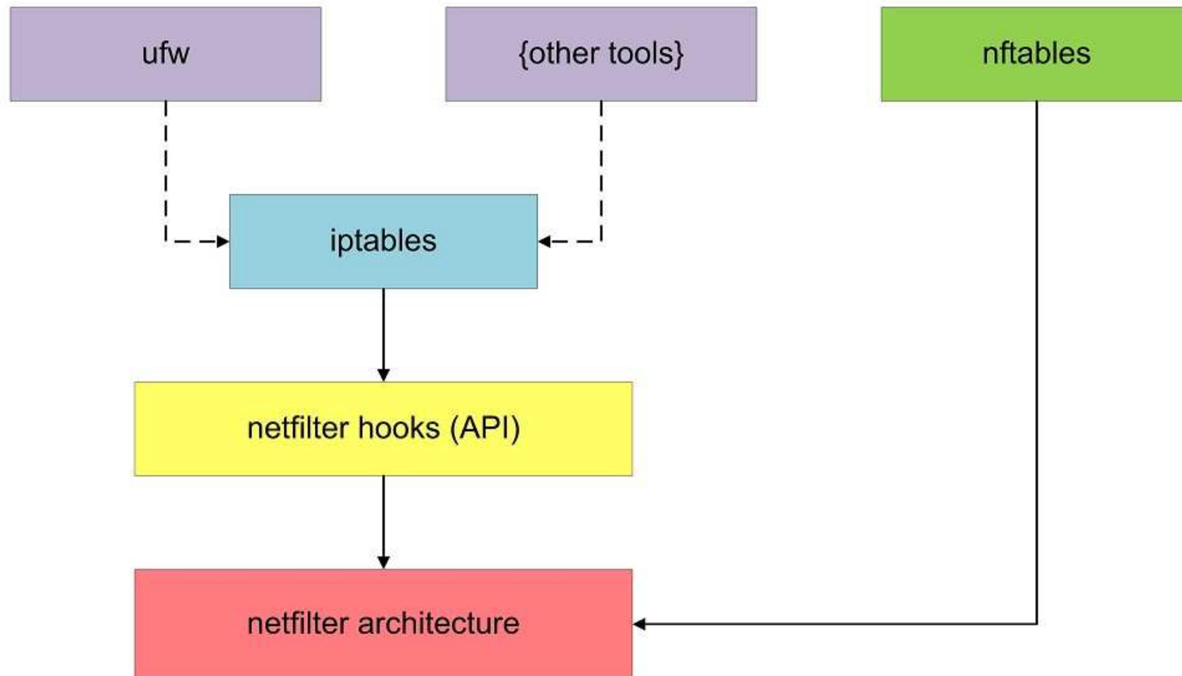

Figure 3: Top Level NetFilter Hooks: NFTables, UFW

While for now, *iptables* remains the de-facto standard, *nftables* may eventually replace it. Regardless of whether or not that ever happens, either may be used. The advantages of *nftables* over *iptables* are:

1. More concise command structure; consolidates filtering to fewer command lines
2. Chains are fully configurable; unlike *iptables*, base chains do not exist
3. Concatenations are supported
4. New protocols are supported without kernel updates
5. Expressions replace and combine the concepts of matches and targets
6. Dual stack IPv4/v6 implementation

Moving from *iptables* to *nftables* is a paradigm shift both intellectually and practically. Prior to *nftables*, Linux' approach to networking has been to embed all controls into the operating system kernel. *Nftables* creates a virtual machine that manages all mangling acts independently upon the networking packets, before presenting the final packet version to the kernel.

There are only two possible paths whereby a future Linux iteration could abandon *iptables* code: a hard fork (remove all *iptables* code), or a "soft" fork where the new system maintains hooks to the old system for compatibility. To date, we currently have the latter. Without a concerted effort to push developers off of *iptables* and onto *nftables*, I hesitate to even call it a 'soft' push. More like an alternative option. Rather, *iptables* and its variants continue to represent the gold standard in Linux networking tools, simply by virtue of

momentum and familiarity (and in spite of their often clunky interfaces or command structures). *Netfilter's* translation code is deeply buried in the Linux kernel. So much so that it will require a major effort to untangle it if the ultimate goal of *nftables* is to be realized (deprecating *iptables*). As I write this content – over five years after the release of *nftables* - the current Linux kernel is 4.19 and there are no signs of the elimination of *iptables* on the horizon.

The *iptables* code is well documented and - in spite of its inefficiencies - it works. Unless someone forces a change, it's easier to continue using 20-year old code than address the task of a monumental re-write, and likely break lots of things until the fact they are broken is discovered and corrected. As the world has become more dependent on networking-based applications such as "cloud computing," the effort required to make an eloquent conversion is greater than ever. To wit, my preceding comment in the introduction: reliability trumps innovation. The Linux networking roadmap is *prima facie* evidence of this design philosophy.

UFW

As if the underpinnings of network packet management in Linux were not confusing enough, Ubuntu introduced another layer of complexity and confusion with the introduction of the **ufw** tool (Uncomplicated FireWall). Originally released with Ubuntu 8.04 LTS in April 2008, **ufw** purported to be, "The default firewall configuration tool for Ubuntu." (UFW, 2017). Claimed to be the "default," yet by default it is disabled. The reality is UFW simply adds another layer; a front-end to *iptables*.

IPv4 and IPv6

Any discussion of networking would be remiss without a brief discussion of IPv6. *Netfilter* handles both IPv4 and IPv6 address translation and routing. Meanwhile, it's split in *iptables*. IPv6 is handled by **ip6tables**, while **iptables** handles IPv4 only. The default routing configuration in Ubuntu has a small amount of IPv6 routing built-in (just a few very broad capabilities that ensure your IPv6 packets don't get blocked by the IPv4 routing process). However, due to the substantial difference in address complexity of IPv6 vs. IPv4, I'm omitting an in-depth discussion of IPv6 routing from this guide, though the basic concepts are the same.

Network Routes

Two systems built-in to Ubuntu manage network routing: **route** (a legacy system) and **ip route**. Both use the same underlying code, yet report networking context a bit differently. **Ip route** has a larger command set. Although **route** has been deprecated for some time, I still find it useful under some circumstances as it provides a different perspective of the routing table not available with **ip route**.

Route and **ip route** view and manipulate network routes. A *route* informs your server where to find other network devices it wants to talk to, based on the destination network address. You can visualize *route* and *ip route* as a system of roadways and *iptables* as a system of traffic lights that regulates the flow of traffic on the roadways. Of course, you could just use **route** and **ip route**, and you'd be able to route network traffic just fine. What you would lose is the ability to fine-tune the traffic's behavior. *Route* and *ip route* control <u>where</u> things go, while *iptables* controls <u>what</u> goes where. *Route* is a system of conduits, while *iptables* filters the data inside those pipes.

To make things a bit more complicated, there is another filtering tool called **ip rules** that is part of the routing system called a Routing Policy DataBase (RPDB). **Ip rules** allows the use of multiple independent routing tables plus filtering algorithms capable of controlling when traffic is filtered and to which routing table. It may be possible to use **ip rules**, not use **iptables**, and accomplish your goals. However, as you'll see later on, **iptables** is more powerful.

Here's a quick-reference diagram demonstrating the relationship between commands and processes.

Concept	Routing	Netfilter
Rules	Yes	Yes
Tables	Yes	Yes
Routes	Yes	No
Chains	No	Yes
Addresses	Yes	Yes

Command	Routes	Filters	Scope
ip route	Yes	No	Routes
route	Yes	No	Routes
ip rule	Yes	No	RPDB Rules
iptables	No	Yes	Netfilter
ufw	No	Yes	Netfilter

Figure 4: Linux Networking Concepts and Associated Tools

And a high-level overview of the process flow.

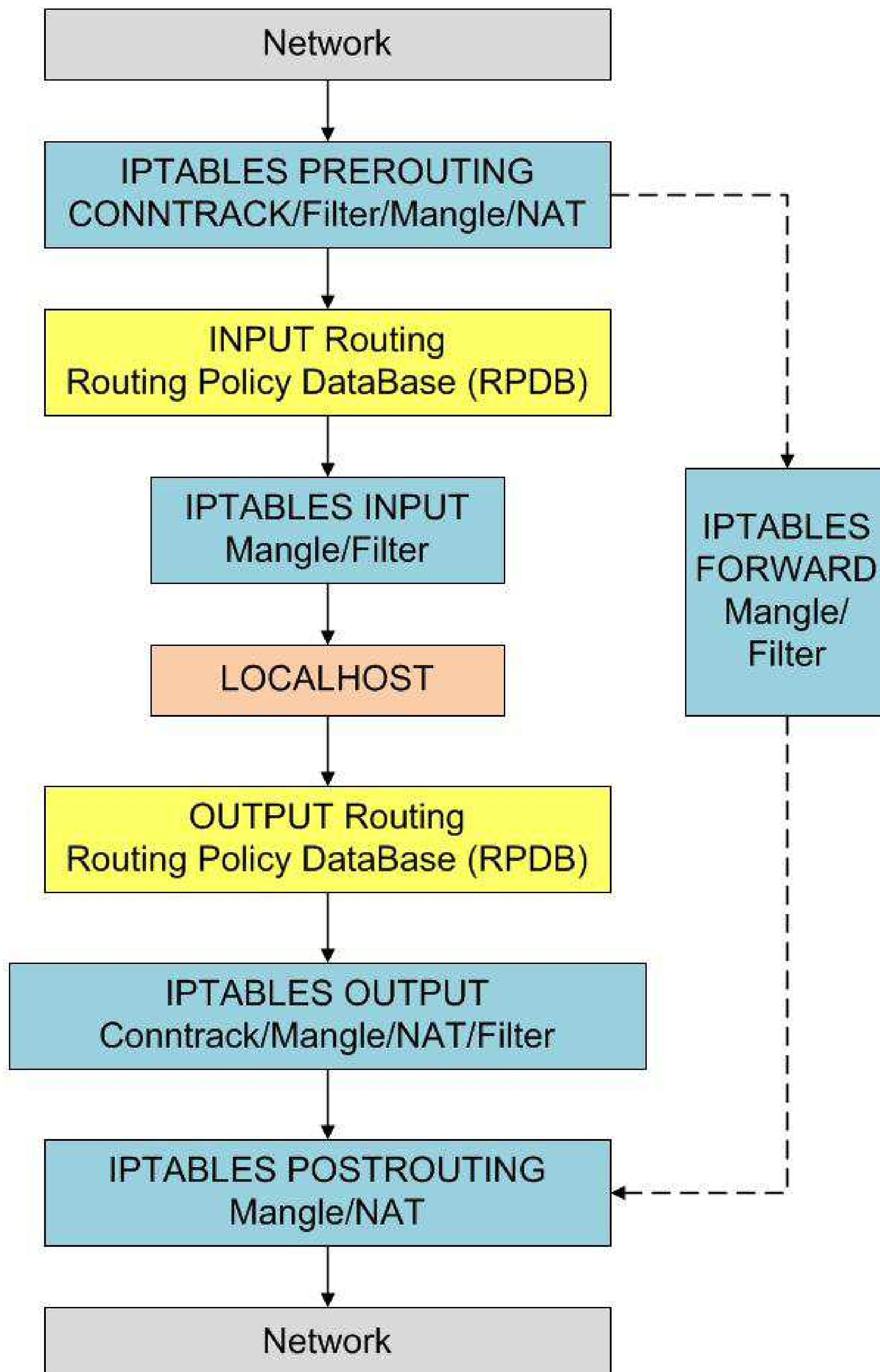

Figure 5: Packet Routing Tree

How Linux Routes Network Packets

netfilter, **iptables**, and **ip rule** have to be some of the most confusing concepts in Linux. Before you begin writing your own packet filtering rules, let's examine the order of operations of network traffic. Breaking down network traffic, which *chains* and *filters* touch each scenario? Below, I've highlighted **chains** in red, **filters** in blue, and **command processes** in green.

Incoming Packets

All incoming packets first traverse PREROUTING...

PREROUTING [RAW table **->** CONNECTION TRACKING **->** MANGLE table **->** NAT table (DNAT) **->** Route Decision**]**

If the packet is addressed to the server, it follows the INPUT chain...

INPUT [MANGLE table -> FILTER table -> SECURITY table -> Local Process]

If the packet is following the INPUT chain path, it's sent to the local host for processing.

Forwarded Packets

Forwarded packets follow the 'incoming' process above, and divert after the PREROUTING step.

PREROUTING [RAW table **->** CONNECTION TRACKING **->** MANGLE table **->** NAT table (DNAT**) ->** Route Decision)]

It then follows the FORWARD chain path...

FORWARD [MANGLE table -> FILTER table -> SECURITY table]

It then gets handed to POSTROUTING...

POSTROUTING [MANGLE table **->** NAT table (SNAT) -> **Out for Delivery**]

Forwarded packets are unique in that both pre and post routing rules are applied to them.

Outgoing Packets

Outgoing packets have a much simpler process. All outgoing packets (that ORIGINATE from your server; i.e. not forwarded through the server) first go through the OUTPUT process...

OUTPUT [RAW table **->** CONNECTION TRACKING **->** MANGLE table **->** NAT table (DNAT) -> FILTER -> SECURITY **]**

The packet is then handed to POSTROUTING... (same process as forwarded packets)

POSTROUTING [MANGLE table **->** NAT table (SNAT) -> **Out for Delivery**]

From there it goes onto the network, routed to the destination host.

The Whole Enchilada in One Picture

Unfortunately, it's quite challenging to find a good diagram of the whole process that does not either leave out important details (such as the Security table hooks) or provides too many details, including other processes not related to iptables.

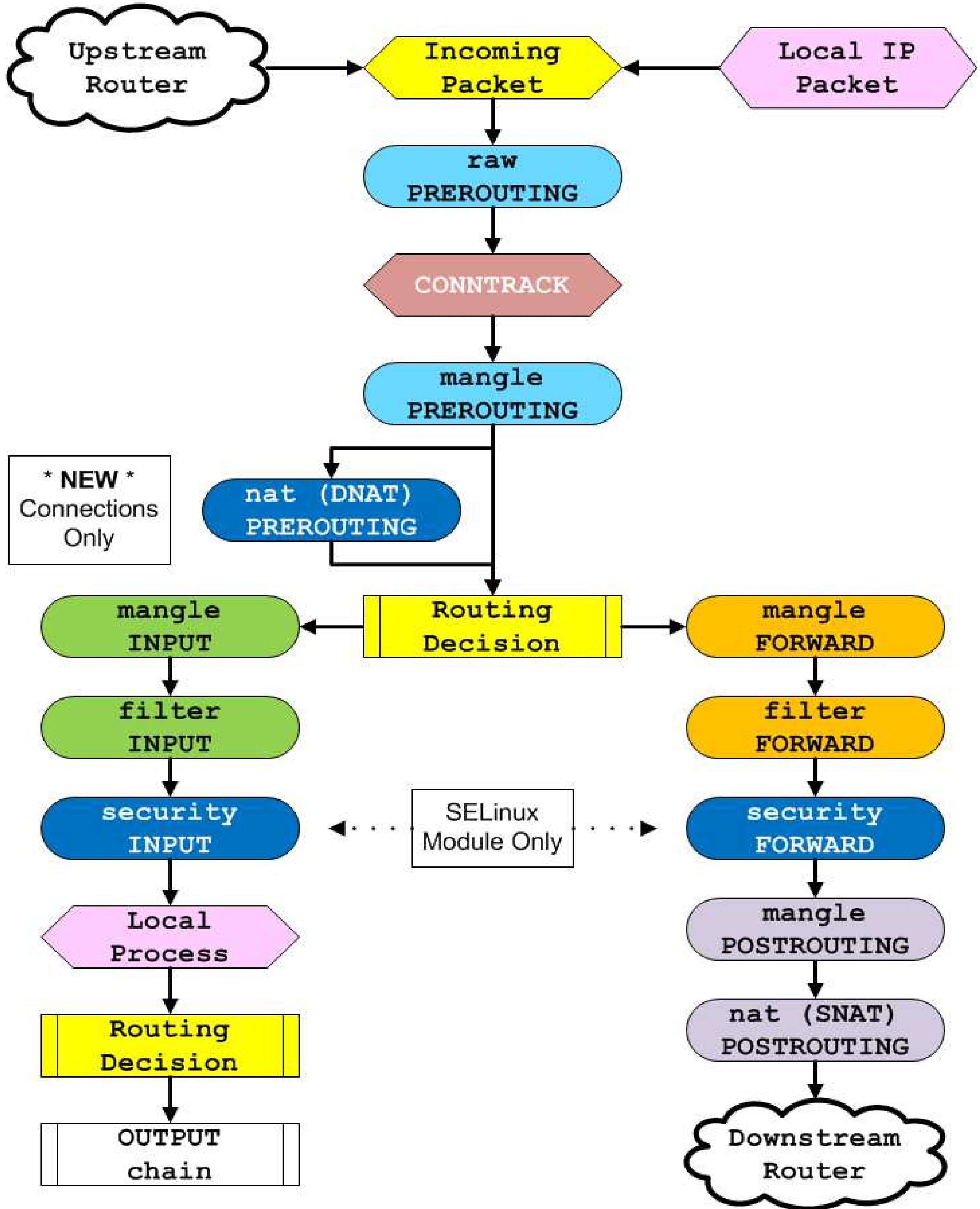

Figure 6: iptables Input and Forward Chains Process Flow Diagram

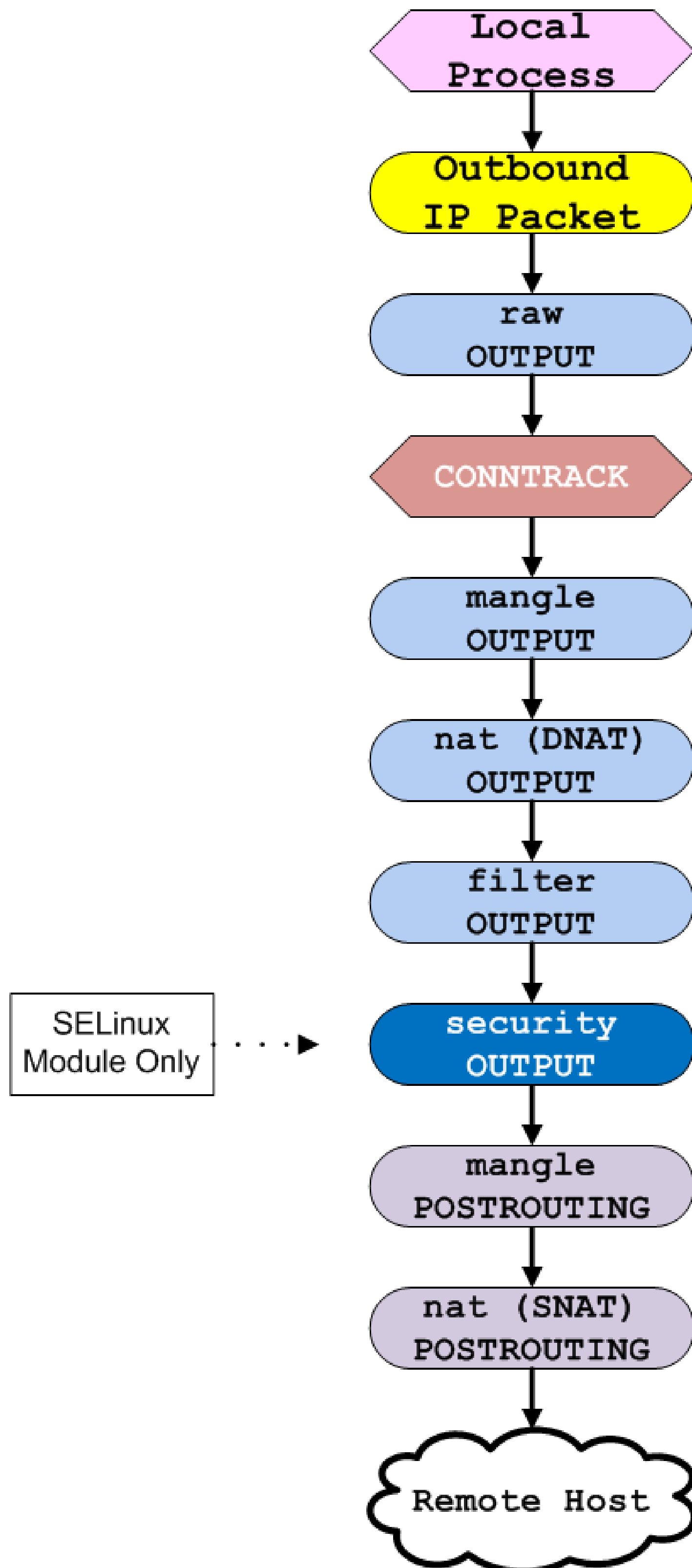

Figure 7: iptables Output chain process flow

Policy Based Routing

Ubuntu stores the server's network route information in routing tables. A *routing table* is a map of a network. Ubuntu uses Linux' policy based routing system, aptly called a *Routing Policy DataBase* or RPDB. The RPDB is a collection of routing addresses, rules, and routes. The "policies" per se are defined by rules. Here's how it works.

A traditional routing table is two-dimensional. It is a single table that includes algorithms (rules) and pointers (routes) to every network connection the server is aware of. Routing a connection is a affair. Apply any applicable rules and off you go. The advantage of a RPDB is two-fold: the separation of rules from routes, and the use of a master routing table index and a collection of tables with the real routing information. This combination allows more granular control over network traffic, such as pointing to a single route from multiple rules, or the opposite (multiple rules pointing to a single table). In short, it allows for improved granular control and segmentation of your network routes.

Another plus of a RPDB is real-time changes can be made to routing tables without interfering with the server's operation. With a traditional single-dimension routing table, you may need to interrupt the server's traffic or have to take it offline in order to modify the table. And if you inadvertently make a mistake while editing a router table, in a single-dimension routing table environment it's much easier to accidentally cripple your server's access. While in a RPDB configuration, isolating changes and creating failsafe traps is easier. In short, it's more foolproof.

The RPDB rule cache is loaded during the server boot-up process. When a routing decision is required, only the cache copy is referenced. What are the characteristics of the RPDB?

- Addresses (source and destination)
- Rules (what goes where?)
- Routes (where's it going?)

In Linux, the RPDB is composed of two databases: a table of rules and a table of routes. Taken together they determine the route assigned to a packet. At a high level, the rules work in conjunction with routing tables like this:

in -> Rule filter -> Rule Match [Highest Priority] -> Routing Table Lookup [Longest] -> *out*

How is a route selected for any given packet?

Rules are examined sequentially, in order of priority. Rules contain a selector and an action predicate. If the current network packet matches the current rule *selector*, the *action* is executed. If the corresponding action is a reference to a routing table (a "look up"), it will point to the index number or name in the RPDB's routing table.

The combined process flow looks like this:

IP Rule Analysis

1. Network packets are processed one at a time
2. The same rules process both incoming and outgoing packets
3. Packets are evaluated against a list of rules, in order of priority, beginning with rule 0
4. First rule matching the packet is executed
5. If the action is a table lookup and the lookup fails, evaluate the next rule

6. If no rule matches the packet, fail with an undeliverable packet error

IP Route Analysis flow

1. The route with the *longest matching prefix* is chosen
2. If more than one match is the same length, the route with the best preference values is chosen
3. If more than one match exists, if ToS is specified, routes that don't match the ToS are dropped
4. If more than one route still exists, then the first ordered matching route in the list is chosen
5. If no matching route is found, the kernel returns to the Rule table and moves onto the next rule

To be clear, in this context "prefix" means the destination IP address, including the netmask (if present).

Let's move on to a discussion of how the main routing process functions and the philosophy behind it. Afterwards, I'll go over the packet mangling process (*netfilter*), and wrap it all together for you with a comprehensive explanation of how these pieces fit together to complete the puzzle.

The RPDB Triad: Rules, Routes, and Tables

These are the building blocks of the Routing Policy DataBase (RPDB) system. Every server has an internal firewall and routing configuration that tells it where to send network packets. **iptables** manipulates packet filters, **ip route** controls routing paths, and **ip rules** controls routing policies on your server.

Before you begin modifying your RPDB, you should be clear on how *ip rules*, *ip route*, and *iptables* are inter-related and work together to manage network packet flow. You may recall in the diagram in the Network Routes section that while all three modules are called at one point or another, which module is called first (**iptables** or **ip rule**) depends on whether the packet is originating from server (new, outgoing packet), an incoming packet, a forwarded packet, or an outgoing packet that's part of an established communication link.

Here's a brief clarification of each portion of the Linux RPDB system.

Addresses

This is pretty straightforward. An address describes the location of a service on a network. Packets must have both source (origin) and destination addresses. A packet's addresses are the most important factor in how it is routed.

Rules

IP rules are pointers that define the location of routes. While their ability to scrutinize a network packet is limited, in combination with routing tables they create a powerful filtering combination.

Routing rules are managed with the **ip rules** command. They are the crux of the Routing Policy DataBase (RPDB) system. Routing rules control how and where traffic flows along routes to a destination address (if it moves forward). Routing rules are effectively filtering algorithms. Each rule contains a *selector* (match filter) and an *action*. They provide instructions to the kernel regarding what to do with the packet.

Selectors may examine one or more of the following parts of the current network packet:

1. Source address
2. Destination address
3. ToS (Type of Service)
4. Incoming interface (that which the packet arrived from)

5. fwmark (firewall mark)

IP rule does not allow filtering based on protocols or transport ports, though that type of filtering is possible via **iptables**, which is discussed later in this guide. For our purposes (basic routing), we are not concerned with the *ToS* value.

The output of each rule is of the following *action* types:

- Jump to specified routing table
- Drop or reject packet
- Return an error
- Modify packet source address

Each rule contains a *selector* and an *action*. The selector is an algorithm or criteria. When the *selector* of the current rule matches the packet, that rule's *action* is executed. Each rule is preceded by an integer from 0-32767. The rule number is its priority. For every packet, the rule database is scanned from highest priority (rule 0) to lowest priority (rule 32767). Each packet is compared sequentially to every rule beginning with rule 0.

IP Rules are explained in more detail in Follow the Rules.

ToS

Type of Service (ToS) relates to the priority of a network packet relative to other network traffic. It is sometimes used in advanced networks to incorporate traffic priority into the datagram of the traffic itself (versus traffic prioritization by other means, such as port, protocol, source/destination address). Traffic shaping and priority manipulation by ToS is beyond the scope of this document.

Routes and Routing Tables

A "route" is a pathway to a destination. It defines the location of a service. A *route* instructs the kernel where on the network to send a network packet so that a packet so that it is one step closer to its destination address. The *route* may be the destination, or it may be a *bridge*, *router*, *gateway, or the localhost*. If the route leads to another device that is not the final destination, the route is called a *hop*.

A "route table" is a collection of routes. You can liken a *routing table* to a map, as they typically contain multiple routes, and it's possible more than one route could lead to the packet's destination. Likewise, a single *route* - a single basic instruction - is like a sign post. It advises the kernel of the next step in a packet's path to its destination. Similar to *rules*, there may be times when more than one route is applicable to a particular packet. When this happens, the <u>longest</u> matching route wins. Practically speaking, the longest route should be the <u>most specific</u> route and it is generally safe to presume that will be the case.

Routes

We will begin the detailed discussion of each component of the RPDB triad with *routes*.

Routes are managed by the **ip route** and **route** tools. The difference is **ip route** is current (*iproute2*) and **route** is the deprecated version (*iproute*). It is however, mentioned several times in this document because it is useful for some things, and the legacy code is still present.

The Master Routing Table

Linux routing table entries are manipulated via the **route** and **ip route** commands. A surprising weakness of both tools is they're only capable of displaying a single route table at a time. I say 'surprising' because up to 256 independent routing tables are possible through the use of an indexed *master routing table*, yet command line tools only display the contents of one route at a time.

Rather than using a single fixed routing table as in a traditional router table configuration, the Linux RPDB system uses the master routing table as an index to refer to route tables indirectly. The beauty of the system is in this ability to segment routing into individual compartments, plus a large database capable of storing up to 32,768 rules that interact with the route tables. The end result is a complex web of routing, though the real workhorse is the rules portion.

The *master routing table* is a file. It contains an ordered list of up to 256 integer and name associations. Each line in the file is a combination numeric and named value pair indexed to a specific router table. This is part of Linux' Routing Policy DataBase (RPDB) system. Rules are utilized to direct the kernel to an indexed router table for processing.

The master routing table is always the same filename, and is always found in the same location:

/etc/iproute2/rt_tables

The file is a simple affair. Take a look at yours.

```
cat /etc/iproute2/rt_tables
```

You'll see a list of the routing table numbers and names that looks something like this:

```
#
# reserved values
#
255     local
254     main
253     default
0       unspec
#
# local
#
#1      inr.ruhep
```

The master routing table contains the list of all the routing tables on your server. The following tables are included in the table list by default on an unmodified system:

- local — Local and broadcast addresses; do not modify or remove
- main — Operated on by **route** and **ip route** processes; default when no policy specified
- default — Reserved for post-processing rules
- unspec — Failsafe; do not modify or remove
- inr.ruhep — Legacy reference; theoretically unnecessary

Note the etnry, "*#1 table inr.ruhep"* is commented out. This is legacy code and may be ignored or deleted.

The master routing table allows you to utilize the RPDB to its full capacity by associating rules with independent routing tables. However, many people don't realize it exists, because Linux' network routing system runs out-of-the-box with no modifications. How? By default, all network traffic is passed through. If you don't have a need to control, filter, or redirect any network traffic passing through it then you really don't have any reason to modify your server's routing table(s) at all.

Default (Built-in) Routing Tables

There are four built-in route tables in Ubuntu: **default**, **local**, **main**, and **unspec**. These routes are all special, though how they are named is a bit confusing.

The *default* table is empty. Contrary to the logic of its name, the *default* table is not used. You may choose to remove it, though leaving it does no harm.

The *local* table handles TCP/IP traffic internal to the server (e.g. between internal ports), and includes the loopback adapter and broadcast traffic.

The *main* table is the primary routing table or what I would call the true 'default' table. Any **route** or **ip route** command where the table is not specified will action the *main* table by default.

The *unspec* table name is shorthand for "unspecified." It is a set of instructions for the kernel to follow when all other routing paths fail.

Now, try this command for each of the three standard router tables (*main*, *local,* and *unspec*):

```
ip route show table {table name}
```

Such as

```
ip route show table local
```

Contrary to what one might expect, the number of each table is irrelevant. Their order in the *rt_tables* file doesn't matter. What is significant are the table names. Each table must have a unique name/ number reference within the table file. This file is simply an index. This is one of the few components of packet filtering that does not require restarting the server for changes to take effect. The routing table index can be modified at any time and the changes will take effect immediately. There is no cached version of this file. It is read every time a table is called by a rule.

If the order doesn't matter, why have an ordered list? First, legacy: this format was created a long time ago, and it's essentially just been stuck in Linux since the kernel 2.1/2.2 era when the RPDB concept was created. Second, it creates a lower boundary of table numbers, ensuring a user who adds a custom table chooses a numeric value in between 1 and 252 (the default table). Since the entire RPDB is cached in memory, if you maxed out the master routing table, the kernel would need to allocate enough memory to store 256 routing tables. Remember, each routing table may contain an unlimited number of routes.

Examining Existing Routes

The next step is to figure out how your RPDB is currently branching network traffic. Remember, this process is a combination of routing tables indexed in the master routing table and rules in the routing rules database. Once you understand how to discover your existing network routes, you'll be able to use these techniques to verify route changes in the future.

You may use several different tools in Ubuntu server to identify routes and hosts on your network (LAN), their outbound interface, and the order in which your server prioritizes its network routes. Examine your current routing structure with the **ip** command set. Run this command and examine the output:

```
ip route list
```

On a fresh server, your output should look similar to this:

```
default via 192.168.10.10 dev eth0
192.168.10.0/24 dev eth0  proto kernel  scope link  src 192.168.10.11
```

Or you could use this command and view the same result:

```
ip route show table main
```

Why is the output the same? The default table in **ip route** is the 'main' table. The first command presumes the default routing table, because a table is not specified. "Default" in the context of the command refers to the routing table called *main*.

What does this output mean?

The *default* route is used when a more specific route cannot be found (thus the term 'default'). This is a little different from the question of which routing table is being displayed. You may now begin to see why Linux routing can be so confusing! When viewing the table output, as in the two lines above, the word, "default" means, "If no other route matches the destination IP address more specifically, select this route."

When you're talking about a default routing table, in Linux that normally means the *main* table. When you're talking about a default selector in a routing rule, to *iproute2* that means the route to be used when a more specific route is not identified. Linux prioritizes network traffic route selection based on a longest match model. If a longer route cannot be identified, the kernel will use the route identified as the *default* route.

Notice in the example above the physical interface is also specified (eth0). A fresh RPDB in Ubuntu is configured such that the default state is to allow all traffic through the server's primary interface to your WAN. In this case the default path is pointing to the upstream router or gateway at 192.168.10.10 on the eth0 interface. This only pertains to *outgoing* traffic. Remember, default means "use this route if no other route better matches the destination address."

The second line shows a route to a network 192.168.10.0/24 via the eth0 interface. The "scope link" indicates this route is directly linked to the server. In other words, it's a LAN. The "src" is the IPv4 source address assigned to packets leaving this server and proceeding along this route (source NAT). Put another way, this line tells your

server there is a local network with a route of 192.168.10.0/24, and when directing outgoing traffic to that network the current server will identify itself with IP address 192.168.10.11.

Though it doesn't provide a wealth of information, this output example identifies the server's connections to the network at a high level. Taken together, those two lines indicate the following facts:

1. There is a network route 192.168.10.0-192.168.10.255 that must be a LAN because this server's source IP address on that route is set to 192.168.10.11;
2. Network packets not destined for the LAN will be routed to 192.168.10.10 on the same interface (eth0) and will not have their source IP address modified by the routing rules;
3. There must be an upstream gateway or router at 192.168.10.10.

IP Route Syntax

This is a good opportunity to explain what some of the words mean that you'll see in the *ip route list* command output.

The following are the relevant *Route Types*:

- **broadcast** - Packets are sent as link broadcast messages
- **blackhole** - Packets are discarded and dropped silently
- **local** - Packets are delivered to the loopback device (local to the server)
- **prohibit** - Packets are rejected; error returned, "Communication is administratively prohibited"
- **throw** - Table lookup terminated; packet is dropped; error message returned, "net unreachable"
- **unicast** - Path to a destination address (this is what most routes are)
- **unreachable** - Packet is discarded and ICMP error message returned, "host unreachable"

These are the most relevant *Control Values*:

- dev [name] — Name of network device to route the packet through (e.g. eth0)
- scope [value] — Scope of area where valid { host|link|global }
- proto — Which process created the route (i.e., is it temporary or permanent)

Scope Values. "Scope" describes where the address is valid.

- global — Valid everywhere (globally); default scope if none specified
- link — Local; valid only on this device
- host — Only valid <u>inside</u> the current host (server)

Proto Values. Also known as "RTPROTO" or "route protocol." It identifies the process that created the associated route.

- boot — Added after boot; temporary
- static — Override added by sysadmin
- kernel — Added by kernel (this is normally what you want to see)

As you can see, one can group several of these into a category of "drop the packet" (namely *blackhole*, *prohibit*, *throw*, and *unreachable*). Dropping and rejecting packets may also be accomplished via **iptables**. The *unicast*

(forward packet to address such-and-such), *local* (send to localhost), and *broadcast* actions instruct the kernel where to send the current packet.

Adding New Routes

So, you've decided you want to add a new route. Perhaps you want to setup a VPN tunnel or add a new network interface to your server and you need to tell it how to use the new interface. Adding a new route boils down to instructing your server how to direct traffic from or to a specific IP address or range of addresses. You either want to create a route the server doesn't understand or see right now, or add a new one that changes how your server currently directs traffic.

Regardless of why you want to add a new route, the process is the same. Just remember routes affect both inbound and outbound network traffic.

There are three options when adding a new route:

1. Add a route to the default routing table (*main*)
2. Add a route to another existing table
3. Create a new route table and add at least one new rules that points to it

To add a new route, the **ip route** command argument structure is:

> ***ip route add*** *[type] [prefix] via [next hop] dev [interface name] table [table name/ID] src [source IP]*

"Prefix" means the destination IP address, including netmask (if present). If "type" is omitted, *unicast* is presumed. Nearly all routes are unicast. For other types, refer to Routes and Routing Tables for information. "Next hop" is the gateway or upstream router the packet will be sent to.

If *prefix* is set to "*default*" it is the same as assigning an IPv4 address of 0/0 or 0.0.0.0/0 or 0.0.0.0/0.0.0.0.

The formatting boils down to the identification of the destination IP address or range, whether the route is direct or via a gateway, network device, and it may contain other values depending on whether or not the route points to a gateway, host, or a group of IP addresses (e.g. a LAN branch). The preceding sections (especially Examining Existing Routes) have more detailed explanations on the syntax.

Here's an example of how to add a new route to a route table called "custom." The new route directs all traffic not routed by a more specific route to a gateway router at 192.168.1.100 via the *eth1* network interface.

```
ip route add default via 192.168.1.100 dev eth1 table custom
```

If you translated this to English, it would read something like this, "Add a new route to the table named *custom* that by default routes all traffic to a router at ipv4 address 192.168.1.100 over device eth1."

You can verify the results of your handiwork with this command:

```
ip route show table custom
```

Warning: All RPDB rules are loaded into the kernel's memory when the server starts up. If you make changes to *ip rules* or *ip routes* and wish to utilize them prior to the next system reboot, you must flush the cache. This forces the kernel to reload the rule and routing databases.

Even though your new routes are now created, they will not be honored by the server until after you reboot or forcibly flush the ip rule and route caches. To do this, run

```
ip route flush cache
```

Best Practices for Routing Tables

Here are some brief concepts for you to keep in mind when adding new routing tables to your master routing table file.

1. Ensure table name and number are both unique!
2. Do not modify default values in the file
3. New table references should be numbered between 100 and 200

Adding New Routes to Default Routing Table (main)

To simply create a new route in the default routing table, you just need to insert a new route in the *main* table. This is accomplished via **ip route**.

First, take a look at the current state of your *main* routing table. Remember, your table labeled *main* is the actual default routing table.

```
ip route show
```

or

```
ip route show table main
```

or

```
route -n
```

If you ever see a route with a non-zero *metric* value, it is a priority value. Metric is an arbitrary 32 bit number that delineates route preferences. Smaller values indicate higher priority. Zero is the highest priority metric. 65535 is the lowest.

Now, follow the instructions below to create new routes in the *main* table. To add a new route to the default *main* table, simply omit the *table <table ID>* portion of each command line.

Adding New Routes to Existing Routing Tables

Adding a new route to an existing table that is not the *main* table is a simple process. Routing tables have a very basic command structure. The possible routing types are described above in Routes and Routing Tables.

For the purpose of this discussion I'm going to focus only on the *unicast* route type. This narrows down the route configurations to the most common scenarios when the server is not a router. Thus, the command format to add a new route becomes:

ip route add {[destination ip/mask] [default]} {via [ip/mask]} {dev} [device] {table} [table ID] src [source ip]

Route Priority Processing

How does *netfilter* determine which route to choose from a routing table? What happens if a packet matches more than one route in the table?

Complex routing tables in large networks can contain hundreds of route entries. When the Routing Policy DataBase (RPDB) is ready to scan a route table and attempt to match a packet to one of the routes in that table, how does it decide which route to apply?

Recall that at this juncture, the RPDB has been directed to the current table by a rule. Routes are compared with the current packet based on the route's *prefix* and ToS value (if any). The *prefix* is a pair of values equal to the destination IP address and its netmask. The RPDB compares the current packet to each route in the table. The most specific match will be chosen, in this order:

1. Destination IP address of the packet matches the prefix, up to the length of the prefix
2. ToS bits of packet and route match
3. The route's ToS=0 (regardless of packet ToS value)

If more than one route matches the packet, the list is pruned in the following order until only a single route remains:

1. Longest matching (most specific) destination IP address including netmask
2. If a single route ToS and the packet ToS bits match, use that route
3. Routes where ToS=0
4. If no ToS=0 routes exist, fail on error (unreachable)
5. If multiple ToS=0 routes still exist, select route with best preference value
6. As a last resort, select the first route in chronological order

IP Route Command Examples

Here are some examples demonstrating how to structure the command line of common scenarios.

Point Destination to Gateway Router

Destination IP address may be single address or sub-net; CIDR (netmask) optional. These instructions direct the packet with indicated destination IP address to the indicated gateway router.

ip route add [destination address/mask] via [gateway IP]
ip route add n.n.n.n/n.n.n.n via n.n.n.n/n.n.n.n
ip route add n.n.n.n/nn via n.n.n.n/nn

Set Default Route to Gateway Router

These instructions assign the default route to the indicated gateway router or next hop.
ip route add default via [gateway IP]
ip route add default via [next hop router IP]

ip route add default via n.n.n.n/n.n.n.n
ip route add default via n.n.n.n/nn

Point Destination to Next Hop

Destination IP address may be single address or sub-net; CIDR (netmask) optional. These instructions direct the packet with indicated destination IP address to the indicated router that is the next hop.

ip route add [destination address/mask] via [next hop IP]
ip route add n.n.n.n/n.n.n.n via n.n.n.n/n.n.n.n
ip route add n.n.n.n/nn via n.n.n.n/nn

Add Route via Interface

Add a new route tied to a specific interface.

ip route add [destination address/mask] dev [interface name]
ip route add default dev [interface name]
ip route add n.n.n.n dev [interface name]
ip route add n.n.n.n/nn dev [interface name]
ip route add n.n.n.n/n.n.n.n dev [interface name]

Set Default Route to Specific Interface

ip route add default dev [interface name]

Policy Routing (Routing Tables)

When applied to the RPDB policy routing model, the syntax is the same with the exception of the table specification. Simply append "table [table ID]" to the end of the command line. "Table ID" may be either the name or number of the table in the *rt_tables* file. Here are a few examples:

ip route add default via [next hop] table [table ID]
ip route add default dev [interface name] table [table ID]

Such as:

Set default route in table "vpn" as the *tun0* interface:

ip route add default dev tun0 table vpn

Set default route in table *test* to gateway at 192.168.1.1:

ip route add default via 192.168.1.1 table test

Automatic Fallback / Down Detection and Redirection

This scenario involves two gateways and how to establish one of them as a priority path while using the other as a fallback. An example would be if your endpoint was attached to two independent internet service providers on the same network interface. How does new outgoing traffic get routed over one versus the other? There are multiple ways to decide. One method involves the use of UP and DOWN route commands in your server's network configuration.

Here is an example of setting up your router table using that method and branches network traffic to one of two new router tables if the destination port matches.

1. Create a new table in */etc/ip route2/rt_tables*. You can name anything you like.

```
echo 1 vpn1 >> /etc/ip route2/rt_tables
echo 2 vpn2 >> /etc/ip route2/rt_tables
```

2. Create new routes that point to the new tables.

For the table *vpn1*:

```
ip route add default dev tun0 table vpn1
```

For the table *vpn2*:

```
ip route add default dev tun1 table vpn2
```

3. Create filter rules. Use *iptables* **mangle** table and the **set-mark** action to map specific port destinations to a particular routing table. The example below branches traffic for destination ports 22 or 80. Notice a different **mark** value is used depending on which table the traffic should be redirected to.

```
iptables -A PREROUTING -t mangle -i eth0 -p tcp --dport 22 -j MARK --set-mark 1
iptables -A PREROUTING -t mangle -i eth0 -p tcp --dport 80 -j MARK --set-mark 2
```

4. Add new ip rules to funnel marked traffic to the new route tables.

```
ip rule add from all fwmark 1 table vpn1
ip rule add from all fwmark 2 table vpn2
```

Source NAT in ip route

Though there is no true NAT capability in *iproute2* (**ip route** commands), it can change the source address of a packet before it has left the local server *(localhost)*. It is a subtle distinction. True NAT was removed from *iproute2* and relegated to *iptables* in Linux kernel 2.6. The RPDB can only change the source IP address when the packet is new and originating from the server (localhost).

You may wish to review the diagram above under the Network Routes section. After leaving the localhost and prior to reaching the OUTPUT *iptables* chain, the RPDB is capable of altering the outgoing source packet. This is technically not NAT, but is a stateless mangling of the packet and only works with new connections. Either way, it is generally discouraged. There are very few scenarios where using the RPDB to mangle a new packet's source IP make sense. It's almost always more desirable to perform source NAT-ing (SNAT) in the *iptables* POSTROUTING chain.

Also note that **ip route** cannot be used to change the destination address of a packet (destination NAT or DNAT). *iproute2* originally included a NAT command, but it was deprecated in the 2.6 Linux kernel in order to encourage the use of *iptables* for packet NAT changes (using stateful SNAT and DNAT). This is discussed in greater detail in the *iptables* section of this document.

While it may appear to be a form of NAT, it is not. The *src* parameter mentioned previously modifies a packet's source address. However, it only works with packets originating from the current host. This is what differentiates it from source NAT (SNAT), which is accomplished via *iptables*.

There are only two circumstances where I'd encourage you to use **ip route** to modify the source address of a packet. The first is when more than one LAN is connected to a single network device. Under that circumstance, it makes sense for **ip route** to mangle the source address instead of *iptables*, because **ip route** knows which outbound route the packet will be taking and the packet is originating from the server. This solves a potential problem where the packet could get dropped because the next (hop) device tries to return a reply to the server, but the packet's source address is not in the correct range. The result is the packet is dropped and the originating server never gets a reply.

The other scenario where the *src* action makes sense in **ip route** is when the server has multiple network devices. In conjunction with specifying a particular network device to send an outgoing packet, it makes sense to use this opportunity to also set the source IP address for that outgoing packet.

Deleting Routes

Deleting routes is the reverse of creating them. You must provide sufficiently specific information that the **ip route** command is able to isolate a single matching rule for deletion. The full route instruction may be required, particularly if your table contains many routes.

ip route delete <remainder of route statement>

Don't forget to flush the cache afterwards.

```
ip route flush cache
```

Creating New Routing Tables

New routing tables in the Routing Policy DataBase (RPDB) are created by simply adding a line to the **rt_tables** file. All that is required is an index value and table name. Each table must have a unique index value and unique name. Beyond that, the chronological order in the table doesn't matter. See The Master Routing Table for more information.

Adding a new route table is straightforward, but do you need to? If you're just adding a few new routes, it may make more sense to simply use the existing *main* table. Ask yourself: should I be adding a new routing table, or simply adding a new route? If you just need to add a new route, the *main* routing table can be utilized. However, if you are considering marking packets (*fwmark*) or creating a conditional branch such as a split VPN, a separate table can be very useful.

When you create a new routing table, you must also create a minimum of one corresponding rule that points to it, and you need to design the routes in the table to handle any possible outcomes. Three steps are involved:

1. Add new entry to master routing table
2. Populate new router table
3. Create rules that point to new table

Best Practices to keep in mind when adding new routing tables:

4. Ensure table name and number are both unique!
5. Do not modify default values in the file
6. New table references should be numbered between 100 and 200
7. Use only lowercase characters for your table name

Step 1. Amend the master routing table. There are two options to do this.

1a. Method #1. Open the master routing table file for editing.

```
nano /etc/iproute2/rt_tables
```

Choose a value between 1 and 252. As an example, let's use 200 for the table number and "stealth" as its name. The order of your tables in the master routing table doesn't matter; rules control the flow. These entries in the master routing table just tell the kernel where to find the route information. Here's how your new table might look:

```
#
# reserved values
#
255     local
254     main
253     default
#
# vpn table
#
200     stealth
#
0       unspec
```

Remember, the order of indexes in the master routing table doesn't matter. Now that you know how to add a new table, let's continue the process of learning how to examine existing routes in detail, how to populate a new routing table, and how to make use of it.

1b. Method #2. Use a command line to append a new line to the end of the master routing table.

```
echo 100 custom >> /etc/iproute2/rt_tables
```

This command will place the new table entry at the end of the file with the name, "custom" and an index or table ID of 100. The order of the router table entries in the file is irrelevant. What is important is to avoid duplicating index numbers, and to avoid using the pre-existing default numbers (0, 253, 254, and 255).

Step 2. Populate your new table with one or more routes.

Now, we still have a problem, which is the fact the *custom* router table you just created is empty. Take a look at the syntax of your existing tables. Routing tables have a very basic command structure.

Here's an outline of how the routes are structured in a table:

> *[destination ip] | {via [ipv4]} | dev {device} | {proto [kernel or static]} | {scope [scope type]} | {src [source ipv4]}*

The formatting boils down to the identification of the destination IP address or range, whether the route is direct or via a gateway, network device, and it may contain other values depending on whether or not the route points to a gateway, host, or a group of IP addresses (e.g. a LAN branch). The preceding sections (especially Interpreting Existing Routes) have more detailed explanations on the syntax.

Here's an example of how to add a new route to your route table. The table will have just one route, which directs all traffic to a gateway router at 192.168.1.100 via the *eth1* network interface.

```
ip route add default via 192.168.1.100 dev eth1 table custom
```

If you translated this to English, it would read something like this, "Add a new route to the table named *custom* that routes all traffic to a gateway at ipv4 address 192.168.1.100 over device eth1."

You can verify the results of your handiwork with this command:

```
ip route show table custom
```

Step 3. Create at least one rule that points to your new table.

This is arguably the most important part of this process. You need to be cognizant of how a new rule you create will behave in the context of other routing rules in the Routing Policy DataBase (RPDB). Remember, the kernel won't pay any attention to your new route table and route unless you have a rule that instructs it to use them. The most important factors when creating a new route rule are:

1. Rules are processed in priority order with 0 as the highest priority;
2. Longest matching rule wins a tie
3. If more than one rule is matching and lengths are identical, the higher priority rule wins

Routing rules are stored in the RPDB and created via the **ip rule** command.

Special Routing Use-Case Scenarios

Force Source Address

The *src* parameter forces outgoing traffic along a particular route to appear to be coming from a specified source IP address. An example of where this could be useful is directing outgoing traffic as desired onto a shared network interface. The format is:

ip route add n.n.n.n/n.n.n.n dev[device] src n.n.n.n

An example:

ip route add 10.10.14.107 dev tun0 table vpn src 192.168.1.100

Which roughly translates to, "Add a new route to table vpn that directs traffic addressed to destination address 10.10.14.107 to change this server's source address as 192.168.1.100 and send the packet over interface device tun0."

The use of *src* only works on packets originating from the server and is generally discouraged. See Source NAT below for more information.

Silently drop packets.

ip route add blackhole [destination addr/mask]

Reject packets with ICMP / "Host unreachable" response.

ip route add unreachable [destination addr/mask]

Reject packets with ICMP / "Administratively prohibited" response.

ip route add prohibit [destination addr/mask]

Reject packets with ICMP / "Net unreachable" response.

ip route add throw [destination addr/mask]

Single Network Connection, Multiple LANs

Here's an example. Let's say you have two LANs accessible from your server via a single network device.

LAN1 has an address range of 192.168.10.0-192.168.10.255 and a gateway at 192.168.10.10
LAN2 has an address range of 10.10.10.0-10.10.10.255 and no gateway

Your server wants to send data to another device at address 10.10.10.14. Let's run the **route** command to see what the routing table looks like:

```
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         192.168.10.10   0.0.0.0         UG    0      0        0 eth0
192.168.10.0    0.0.0.0         255.255.255.0   U     0      0        0 eth0
10.10.10.0      0.0.0.0         255.255.255.0   U     0      0        0 eth0
```

Examining the **route** command output, you see there is a gateway at 192.168.10.10 and two local routes (192.168.10.0/24 and 10.10.10.0/24). So, what's the big deal?

The issue is guaranteeing outgoing packets sent from this server to each LAN will have responses returned back to your server. We know the longest matching route prefix will be chosen to send the packet to 10.10.10.14. However, we don't know if the source address will indicate it's coming from the 192.168.10.0/24 network or the 10.10.10.0/24 network. We obviously want it on the latter, because we know there is no gateway on the 10.10.10.0/24 network, which means any address outside the network's range will fail because no other server on that network will respond to the packet.

The problem is from this display, we don't know if that guarantee exists or not. Did whomever setup this routing table include the *src* parameter? We can't tell. This is where the **route** command is lacking. We need more detail. Instead, try running **ip route list**. And you get this display:

```
default via 192.168.10.10 dev eth0
192.168.10.0/24 dev eth0  proto kernel  scope link  src 192.168.10.11
10.10.10.0/24 dev eth0  proto kernel  scope link  src 10.10.10.11
```

Wow. Now we can see there is a *src* parameter for both LANs, which is excellent. That means depending on which LAN a packet is addressed to (destination), we know it will come back to this local machine because it will leave this server with an address on the same LAN it went out on.

Multiple Network Connections, Multiple LANs

A more common scenario is where you have multiple network devices on a server and each is connected to an independent network. In this case the *src* parameter serves a similar function: guaranteeing which ip address is specified for an outgoing packet that is part of a new connection initiated from your server. By using *src* to force a particular source IP address for a particular outgoing route, you ensure the return packet will find its way back to your server.

Let's briefly examine how the **ip route** commands might look if you were setting up a table called test to do this. Let's presume you already created the *test* table and can use that for testing purposes.

```
ip route add default via 192.168.10.10 dev eth0 table test src 192.168.10.11
ip route add 192.168.10.0/24 dev eth0 table test src 192.168.10.11
ip route add 10.10.10.0/24 dev eth0 table test src 10.10.10.11
```

You can see each route specifies the source IP address when a packet traverses it. This ensures the packet will be able to return to this host regardless of its destination.

The Route Command: An Oldie But a Goodie

What if you'd like more granular information about your network routes? Although it has been deprecated for some time, the **route** command can be useful in getting a different perspective. I find it helpful for illustrative purposes and (sometimes) in troubleshooting. Unfortunately, because it was created prior to the invention of policy based routing, **route** only displays the *main* routing table. Therefore, if you have custom named routes or plan to add them, you won't be able to view them with the **route** command.

The advantage of **route** versus **ip route** lies in its more granular representation of routes. Both commands provide the same information, but **route** makes a few characteristics clearer, such as which routes point to gateways.

To examine the main routing table using **route**, run this at the command line prompt:

```
route -n
```

It produces output similar to this:

```
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         192.168.10.10   0.0.0.0         UG    0      0        0 eth0
192.168.10.0    0.0.0.0         255.255.255.0   U     0      0        0 eth0
```

Notice the values expressed here match what you saw from **ip route**, but now we have more details and they are presented in a consistent format. Each route comprises its own line in a table. Some people find this method easier to interpret and follow the paths. Even if you don't find it easier on the eyes, the fact is there's more information displayed here, such as *Flags* that clarify what things are.

Any entry in this table represents a network route, and you can more clearly identify what is a single device (*host* or *gateway*) versus what is a *route*. Although you don't see the server's own IPv4 address, you do get a clearer picture of what's what based on the *Flags* column. Let's begin with the most common flags.

- G Gateway Destination IPv4 address is a gateway (router)
- H Host Route to a specific network address
- U Route is Up Route is currently active

Although the *Flags* column provides more information, it's the combination of four of these columns that allow us to definitively identify what is what. Let's break it down. Take a look at the first line.

```
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         192.168.10.10   0.0.0.0         UG    0      0        0 eth0
```

This is the upstream router from the server. How do I know that? The 'Destination' and 'Genmask' columns are both equal to 0.0.0.0. This is the same thing as the first line in the output from *ip route list* that showed a 'default' route. If you were to translate that line to a human-readable format, it would be "*Route all traffic*

addressed between 0.0.0.0 and 255.255.255.255 to the gateway at 192.168.10.10 via the eth0 network interface." When both Destination and Genmask are set to 0.0.0.0, it means all traffic will be passed to that route unless a more specific route is identified in the table.

The second line is indicative of the LAN our server is connected to. It instructs the kernel to send any traffic with a destination address in the LAN's IP address range (unicast) directly to that address, out via the indicated interface. Likewise, if a broadcast message is called for it would be sent out over the indicated interface.

```
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
192.168.10.0    0.0.0.0         255.255.255.0   U     0      0        0 eth0
```

How do we know? The 2nd line's flags only include an *Up* flag. The absence of any other flags mean this is a network route connected directly to the specified interface. In this case, it's our LAN as that's the only thing it can be. How do I know? I can deduce this because I already know: 1) This server is on a LAN; 2) The gateway is within the same IPv4 address range; and 3) If I review the *ip route list* output again, I can find this computer's IPv4 address on the network after the *src* parameter.

If you were to translate the 2nd line to a human-readable format, it would be something like, "*Route all traffic addressed between 192.168.10.0 and 192.168.10.255 via the eth0 network interface.*" Basically it means your server's routing table believes anything in that range is connected directly to the specified network interface. This informs the kernel there are no other hops required to reach an address in that range. However, a route doesn't know about hosts (if any) that are on the route.

Did you notice the similarities between entries from the sample tables produced by **ip route** vs. **route** commands? They both present the same information in different ways. One advantage to **ip route** is it identifies the current computer's IPv4 address (src=) on the LAN.

Notice all the table entries point to the same outbound network interface (eth0). Things get tricky when there's more than one network interface. For now, here are some rules you may use when figuring out what each entry in your router table does:

1. If you don't specify a particular table, you will only see the contents of the *main* routing table.
2. Routes are not prioritized in a top-down pattern as you might expect. They are evaluated on a longest-match principle. The routing table will isolate routes where the route prefix matches the packet prefix. It then chooses the route with the longest matching prefix, regardless of the order in which the routing instruction appears in the table. In layman's terms, this translates to selecting the most specific route. I'll explain what the *routing prefix* is in a moment.
3. A line containing a *G* flag and *Gateway* address, but both Destination and Genmask are 0.0.0.0 is the default gateway/router for the specified interface. Outbound traffic directed to the main routing table not captured by one of the other routes will be sent to this gateway.
4. You can have only ONE gateway per interface. Split gateway entries are possible (a gateway appears on more than one line, but has its full address range split into multiple line entries). It is a complex topic that is covered under Split Gateways.

5. A Host can also be a Gateway. However, if you see an entry with both a Host and Gateway flag indicates there is something special about that entry. It could be a router, gateway, or the local gateway for a VPN (Virtual Private Network).

What is a routing *prefix*? The *prefix* is a string consisting of the combined IPv4 address and netmask. The longest prefix matching process effectively means the most specific route will be chosen, though the route may be no different from others in the table. If more than one route is the same length, the route with the highest matching bit will be chosen.

For example, presume you have a packet with a destination IPv4 address of 192.168.10.1 on the eth0 interface. Your routing table rules were entered like this:

```
ip route add default via 10.10.0.3
ip route add 192.168.10.0/24 via 10.10.0.4
ip route add 192.168.10.0/25 via 10.10.0.5
ip route add 192.168.10.1/32 via 10.10.0.6
```

The **route** command may be used to examine your *main* table, which looks like this:

```
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
default         10.10.0.3       0.0.0.0         UG    0      0        0 eth0
10.10.0.0       0.0.0.0         255.255.255.0   U     0      0        0 eth0
192.168.10.0    10.10.0.4       255.255.255.0   U     0      0        0 eth0
192.168.10.0    10.10.0.5       255.255.255.128 U     0      0        0 eth0
192.168.10.1    10.10.0.6       255.255.255.255 U     0      0        0 eth0
```

It seems our packet matches all the rules. Which rule will apply to the packet? Where will the packet be routed? The genmask 255.255.255.255 rule will apply, because the combination of that route's IP address and genmask will have the highest matching bit. It just so happens that is also the most specific route. This is why the longest route is normally the most specific route. What happens if there is a tie? The exact rule filtering logic is discussed later in this guide.

NOTE: The terms "Genmask" and "Netmask" are interchangeable. *Netmask* is an industry-standard nomenclature, while *Genmask* only appears in output from the original **route** tool.

Netstat

Another way to produce the same information shown above is with **netstat** (deprecated).

```
netstat -r
```

You'll see the exact same output you got from ***route -n***. Without the *-n* switch, you'll get the same table except 0.0.0.0 addresses will be replaced with the asterisk (*) and any recognized hosts will be replaced with their FQDN (Fully Qualified Domain Name).

Here's an example output:

```
Kernel IP routing table
Destination     Gateway         Genmask         Flags   MSS Window  irtt Iface
default         pfSense.skynet  0.0.0.0         UG        0 0          0 eth1
172.16.11.0     *               255.255.255.0   U         0 0          0 eth1
```

As you can see, the Gateway connection in this example now has a name instead of an IP address, and the route representing the LAN just has an asterisk under the 'Gateway' column. This can facilitate making these tables easier to read as it becomes very clear that the default traffic path will take your network packets to the gateway with the name provided and that you have a LAN. Which approach is best is personal preference. If you have a split route, this display may end up being more confusing than helpful. Split routing is uncommon except with VPN configurations. I'll go over what it is and how to implement it in a moment, but first it's important to fully understand the route type flags.

Flags: Hosts, Gateways, and Routes

Flags in *route* or *netstat* provide you information about the function of the route. Is it up (connected)? Is it a gateway? Is it another host? What do those things mean?

The three most common network connection types are: *Host*, *Gateway*, and *Route*. Many routed connections refer to a destination that's either a host or gateway. What is the difference? A *Host* is a stand-alone device with a specific address on the network. A *Gateway* is an upstream router that connects you to another network and performs address filtering. Gateways must also have a specific address that points to a single host IP address (/32). A *Route* is a pathway. Think of it like a highway. It's not a destination, but facilitates a packet reaching a destination. *Routes* are always a range of addresses.

Linux adds your LAN to the routing table automatically when the server boots up. It creates a default table consisting of what it detects during the start-up process. At a bare minimum, this normally consists of your LAN and gateway router. While techniques available to override this process, it is strongly recommended you do not attempt to do so.

Split Gateways

A "split" gateway is a technique that allows the routing table to point the same IP address range to two different gateways. A special variation of this technique allows "splitting" the default gateway.

By definition, there can be only one *default* route, but using a variation of the split gateway method, it is possible to trick the routing code into allowing multiple default paths.

Typical Gateway Routing Example

Let's use an example to illustrate. Suppose you have a simple network with two entries in the routing table. One is the route for the local network, which tells the kernel which IP addresses are directly reachable. And the second is the "default gateway," which tells the kernel that in order to reach the rest of the Internet, traffic should be sent to the gateway. The gateway should be a router or firewall device.

```
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
192.168.1.0     0.0.0.0         255.255.255.0   U     0      0        0 eth0
0.0.0.0         192.168.1.1     0.0.0.0         UG    0      0        0 eth0
```

Here you see the top line in the routing table (output above derived via **route -n** command) is a network path with the U (Up) flag set. We can surmise this must be the LAN. The Destination of 0.0.0.0 is equivalent to *default*. If you viewed this routing table using the **route** command only (no switches), you would see this output:

```
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
192.168.1.0     *               255.255.255.0   U     0      0        0 eth0
default         192.168.1.1     0.0.0.0         UG    0      0        0 eth0
```

One or the other can be easier to read, depending on your preferences. In the output above, notice the gateway IPv4 address is replaced with an asterisk (*). This simply signifies there is no gateway associated with the route

on that line. It's just a different way of representing the same thing. Likewise, you now see "default" as the destination for the gateway (G flag).

Either way, there’s only one line comprising the entire *default* route path. The 2nd line in both examples tells us:

1. This route represents the default network path
2. The default path points to a gateway (router)
3. Any packet not already captured by a more specific addressable route will be routed to the default route

There are two crucial pieces of information to note:

1. If a route represents a gateway, it will have an IPv4 address under the *Gateway* column AND the **G** flag must be set under the *Flags* column
2. Both the *Destination* and *Genmask* columns for the gateway are set to 0.0.0.0

Why?

Genmasks and Destination Address Filtering

The *Destination* and *Genmask* column of IPv4 values are combined to form an address range. The *Genmask* is a mask applied to the *Destination* address using a logical AND operation at the bit level. When a particular destination address falls within this range, it is routed to the *Gateway* address (notwithstanding the fact the most specific route will be chosen for any particular packet).

Without getting into the weeds of the bitwise AND binary operator, the easiest way to think of how they work together is to remember a *Genmask* octet of 0 (all bits of the octet byte set to zero) means, "do not change the corresponding destination address octet," while 255 (all bits of the byte set to 1) is the opposite and means, "restrict the range to only the number presented in the corresponding octet of the destination address."

Here are some examples to help ensure you understand the logic:

- Genmask 255.255.255.0 restricts the range to the first 3 octets of the destination address to only the octets specified in the destination value, while the 0 in the 4th octet indicates the range of this particular routing filter is between the 4th octet of the destination and the number 255.
- Using the AND operator to combine 0 AND 255 results in 0, while 255 ANDed with any number will preserve only the bits in the other number.

In this case, the *default* value is 0.0.0.0 under *Destination* and the *Genmask* is also 0.0.0.0. Together these create a range of 0.0.0.0 to 255.255.255.255, or in other words, "every address."

A tutorial on the AND logical bit operator is beyond the scope of this guide, but here is an image[4] that may help with visualizing the concept:

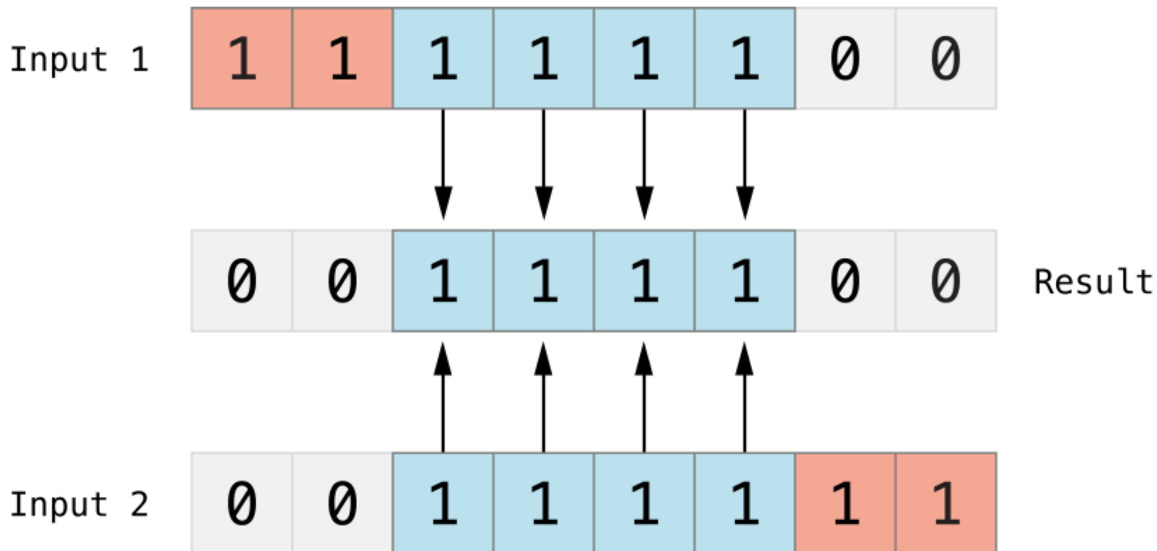

Figure 8: AND Bit Logic Illustration

Recall that the most specific route will always be used, so this will only route traffic to the associated gateway address if there isn't another more specific route to the destination (such as a LAN address).

How To Split a Gateway Route

So now that the logic of a typical gateway route has been explained, how does one create a split gateway? And what is a *split gateway* anyway?

A "split gateway" is perhaps better defined as a *split routing table*. The premise is to route traffic to two or more default routes, depending on whether or not a particular route is up (active). An example would be if you have two different network interfaces and you want all your traffic to pass through one of them as your primary WAN or internet connection, and use the other network adapter if the primary connection fails (down).

Normally, the kernel won't allow you to specify a default route on different interfaces, and it won't allow multiple *default* routes. Regardless of how many network interfaces you have, there can be only one in a single routing table. So, the trick is to split your default route into at least two paths. There is actually an example of this technique a few pages above, under the introductory section on the **route** command.

Let's take another look at the example from the preceding section on the **route** command.

```
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
default         10.10.0.3       0.0.0.0         UG    0      0        0 eth0
10.10.0.0       0.0.0.0         255.255.255.0   U     0      0        0 eth0
192.168.10.0    10.10.0.4       255.255.255.0   U     0      0        0 eth0
192.168.10.0    10.10.0.5       255.255.255.128 U     0      0        0 eth0
192.168.10.1    10.10.0.6       255.255.255.255 U     0      0        0 eth0
```

If you examine the last routing table example, you'll note there's a split route to sub-net 192.168.10.0/24.

Here it is again (excerpted):

```
192.168.10.0     10.10.0.4        255.255.255.0   U     0      0        0 eth0
192.168.10.0     10.10.0.5        255.255.255.128 U     0      0        0 eth0
```

Notice the destination IP addresses are the same, and they are not specific to a single IPv4 address, but rather represent a range (in this case x.x.x.0 or /24 or a 255 address sub-net). However, the Genmasks are different on each line. This is the key. While initially it appears as if there are duplicate entries in the routing table (not allowed), the fact is they are not duplicates because the *Genmask* is different. The Genmask becomes the filter; in this case splitting the /24 address range into two halves (x.x.y.0 through x.x.y.127 and x.x.y.128 through x.x.y.255).

If the packet destination is between 192.168.10.0 - 192.168.10.127, the packet will be routed to a gateway at 10.10.0.4. If the packet destination is between 192.168.10.128 - 192.168.10.255, the packet will be routed to a gateway at 10.10.0.5. Of course, it's not the same thing as splitting the *default* route, but you get the idea. This is what you want to replicate.

The examples above are a bit more complicated, because they also include a 3rd gateway in the split that routes traffic bound for 192.168.10.1 to a gateway at 10.10.0.6. This is a complex scenario that you're less likely to encounter versus a 2-way split. Also notice all these gateways are reached via the same interface (eth0). A common scenario for split gateways is when some traffic is routed to another interface, such as a VPN or another physical network. For example, a large corporate network where depending on the packet destination on the network, the network administrators might choose to route the packet to various different internal gateway routers.

Default Routes

Every routing table needs a *default* route. The *default* route must be a gateway, and it must route all network traffic. When the routing code attempts to match a packet by destination address in the routing table, if there is no more specific route for the packet it will be sent out via the *default* route. On a server that is not a router or firewall, you'll normally have a *default* gateway that is represented by a single line in the main routing table, like this:

```
Kernel IP routing table
Destination      Gateway          Genmask         Flags Metric Ref    Use Iface
default          10.10.0.3        0.0.0.0         UG    0      0        0 eth0
```

Or depending on which command you use it might look like this:

```
Destination      Gateway          Genmask         Flags Metric Ref    Use Iface
0.0.0.0          10.10.10.3       0.0.0.0         UG    0      0        0 eth0
```

A *split gateway* simply splits the default route using the genmask, like this:

```
128.0.0.0       172.16.1.1      128.0.0.0       UG    0      0          0 eth0
0.0.0.0         172.16.1.1      128.0.0.0       UG    0      0          0 eth0
```

Note the gateway is the same on each line. Ordinarily, this would be impermissible, but it works because the combination of Destination and Genmask on both lines creates a split where no possible address may be duplicated by the logic in both lines. The first line perfectly covers the 2nd half of all possible addressable IPv4 addresses in the range between destination and genmask, and the 2nd line covers the first half.

Multiple Default Gateway Routes

We've now covered the concept of split gateways, which basically just means splitting an IP address range so that a portion of it is routed to one gateway and another portion is routed to one or more other gateways. But what if you want to split the *default* route? Since you can't have more than one default route, will it work? Yes. Yes it will. The same logic applies.

Split Gateway with OpenVPN

A common split default gateway scenario is when a server branches some traffic to a VPN. For instance, OpenVPN - a free, open source VPN client/service platform - will modify your routing table by default to direct all outbound traffic over itself. Here's an example of a routing table after an OpenVPN client has been installed on a server. Imagine you see the lines below as a portion of the output from running the **route** command on your server:

```
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         10.150.1.5      128.0.0.0       UG    0        0      0  tun0
0.0.0.0         172.18.10.1     0.0.0.0         UG    100      0      0  eth0
```

Notice how the *default* gateway (destination 0.0.0.0) is split between two lines. The first line directs packet traffic not captured by a more specific rule to a gateway at IPv4 address 10.150.1.5 over the *tun0* network interface, but the line's *genmask* restricts that path only to IPv4 addresses from 0.0.0.0 - 128.0.0.0. The second line directs traffic not captured by a more specific rule to a gateway at 172.18.10.1 if the packet's destination IPv4 address is between 128.0.0.1 and 255.255.255.255.

This is not the only way to accomplish split gateways or split routing. It is also possible (and cleaner) to use discrete routing tables and *ip rules* instead. I have described this technique above for reference purposes and in case you cannot or do not wish to use multiple routing tables.

Follow the Rules

Rules are the lifeblood of the Routing Policy DataBase (RPDB). Rules are processed prior to routes. In order to get to a route, a packet must be directed there by a rule. Rules direct traffic to a specific action or routing table. This is why you'll notice rules use the terminology "lookup" and then reference a routing table. This is because the rules effectively do "lookup" or read and execute the contents of the directed routing table name. For example, the default Linux rules allow all traffic and direct both incoming and outgoing traffic and 'lookup' the default routing tables (e.g. localhost, *main*).

> **WARNING**: Routes and rules are not automatically available during the current session. To activate new rules and routes created with *ip rule* and *ip route*, you must flush the cache; either by rebooting the server or forcibly flushing the cache. To activate new iptables rules that are persistent between server reboots, the process is a bit more involved and is explained in Making Your Routes and Rules Persistent.

New rules are created with the **ip rule** *add* command.

IP Rule Syntax

Like almost everything in this guide, I'm not going to delve into every possible parameter and switch in **ip rule**, but I shall go over the most common variants. The syntax of a rule looks like this:

ip rule add {prefix} {selector} {predicate}

Sounds simple enough, eh? But, what is a *selector*? What is an *action*?

The rule type *add* - as its name implies - adds a new rule to the RPDB. Within the context of the **ip rule add** command there are three components: *prefix*, *selector*, and *predicate*. The *prefix* determines if the rule applies to incoming or outgoing traffic. The *selector* is the filter or what conditions are being applied. The *predicate* is the object of the *selector or an action*.

1. Prefix: from {addr} | to {addr}
2. Selector: to {addr} | *priority <#>* | *fwmark <fwmark>* | *iif <name>* | *oif <name>* | tos <value>
3. Predicate: {blackhole | prohibited | unreachable} |lookup <table id>

It's helpful to break these down a bit further.

The *prefix* is a combination of a *from* and/or *to* statement, and a single IP address or range of IP addresses, with our without a netmask.

- **from** address or range of addresses with the given source IPv4 address or range*
- **to** address or range of addresses with the given destination IPv4 address or range*

* Note: "all" is an acceptable wildcard, which means "all addresses."

A *prefix* of *from* pertains to an incoming filter where the associated IP address is a source address/range, with or without a netmask. A prefix of *to* pertains to an outgoing filter where the associated IP address or range is a destination address/range. These source and/or destination addresses are used to determine whether or not the current rule up for evaluation should be applied to the current network packet.

After the *prefix* comes the next step in the filtering process; the *selector*. Multiple *selectors* may be used in the same rule. These are filtering screens. The *selector* is comprised of the following choices.

- **fwmark** a means of marking packets; decimal values are converted to hexadecimal
- **iif** in-bound interface <name>
- **oif** out-bound interface <name>
- **priority** rule # you wish assigned; must be unique; lower # = higher priority
- **tos** ToS=Type of Service; rarely used by home users; for more info see ToS

The *predicate* is a bit easier to follow. There are four possible outcomes:

- *lookup* return route found in the referenced routing table
- blackhole drop packet silently
- prohibited reject packet and error, "Communication is administratively prohibited"
- unreachable drop packet and return error, "network unreachable"

A Brief Word on *fwmark*

ip rule cannot create or add a mark to a packet. It can detect the presence of an *fwmark* and apply branching logic (filtering) if an *fwmark* exists.

IP Rule Examples

Here are some examples to demonstrate the concepts discussed above.

An example that points to a new table called "*custom.*"

```
ip rule add to 192.168.1.200 lookup custom
```

Let's pick apart this example. The "to" prefixing an IP address means it is a destination address. "Lookup custom" means the corresponding action is to lookup or read the *custom* routing table. So, an English reading of the corresponding new rule would be something like, "If the current network packet destination address is 192.168.1.200, then lookup and execute a route in the table named *custom*."

Here are more examples of rules you could create:

```
# match a source network and direct its traffic to a particular table
ip rule add from 192.168.1.0/24 lookup mytable

# prohibit traffic from a source network
ip rule add from 192.168.1.0/24 prohibit

# blackhole any traffic headed for a particular sub-net
ip rule add to 10.10.10.0/24 blackhole

# direct traffic addressed from a specific address to a specific sub-net
ip rule add from 192.168.1.33/32 to 192.168.2.0/24 lookup default

# direct packet from specific IP to specific sub-net to table named test123
ip rule add from 192.168.1.33 to 192.168.1.0/255.255.255.0 lookup test123

# silently drop any traffic attempting to reach between two sub-nets
ip rule add blackhole from 192.168.1.0/24 to 172.16.11.0/24

# direct marked packets to table test123
ip rule add from 192.168.1.0/24 fwmark 1 lookup test123

# set priority and filter specific addr to specific addr
ip rule add priority 22 from 192.168.1.33 to 192.168.1.200 lookup default
```

Rule Priority

Rule priorities are processed in order from lowest to highest number. Take a look at your current rules.

```
ip rule show
```

An unaltered, new ip rule table will look similar to this:

```
0:      from all lookup local
32766:  from all lookup main
32767:  from all lookup default
```

Rules are evaluated in order against packets based on the *longest match principle*. It means the rule matches the current packet and is also the longest character length rule in the RPDB that meets the matching criteria. In the case of a tie, when more than one rule both matches the logical criteria and is the longest matching length rule and is of equal length as another matching rule, only then does *priority* come into play. At that point the *highest priority match* wins the battle. Functionally, because it applies to IP addresses, this model normally equates to a model of *most specific*, because logically that is going to be the longest matching bit pattern. However, that is not always the case as the length of the rule (in bytes within the rule table) influences the outcome. Therefore, if you have more than one potential matching rule for a particular scenario, it's a good idea to place your preferred outcomes at lower rule numbers (higher priorities), just in case.

Another important concept is if you add a new rule via the command line without specifying the rule number, the next sequential rule number preceding the last used rule number will be selected. For example, if the previous rule you created has a priority of 100, and you then add another new rule but don't specify a priority number, the number 99 will be assigned to the new rule.

To re-cap, the process works like this:

Rule filter -> Rule Match [Longest + Highest Priority] -> Routing Table -> Most Specific Route

Rules filter the packet first, which identifies the appropriate routing table. Rules are evaluated in order. They are numbered from 0-32767. The routing table is then scanned for a matching route based on the source or destination IP of the packet.

Let's break down how these components interact with one another to derive at a route for each packet. Linux evaluates routing requests in this order:

1. Process the rules in order starting with rule #0
2. Check each rule in the database to see if it matches the packet
3. If there is more than one match, choose the longest matching rule (presumed to be the most specific)
4. If no matching rule is found, return a non-reachable error

Take a look at the incumbent rule set after installing Ubuntu. You can see it's very simple and contains just three rules.

```
0:      from all lookup local
32766:  from all lookup main
32767:  from all lookup default
```

If you were to create a new rule and not specify a priority for it, such as this rule:

```
ip rule add from all iif eth0 lookup main
```

Your RPDB table would then look like this:

```
0:      from all lookup local
32765:  from all iif eth0 lookup main
32766:  from all lookup main
32767:  from all lookup default
```

Incidentally, the new rule above would instruct the kernel to route all traffic coming from the incoming interface *eth0* to the *main* router table.

Don't forget to pay attention to the order of your rules and the order of when/how each table is called.

If your server doesn't have custom tables yet you may view your current RPDB rule collection using this command:

```
ip rule show
```

All RPDB rules are loaded into the kernel's memory when the server starts up. If you make changes to *ip rules* or *ip routes* and wish to utilize them prior to the next system reboot, you must flush the cache. This forces the kernel to reload the rule and routing databases. To do this, run:

```
ip route flush cache
```

How Default RPDB Rules Function

While reading the Rule Priority section above, you might have wondered how the default *ip rules* can possibly be useful. After all, they are simply three "from" rules. Let's review them.

```
ip rule show
```

Displays:

```
0:      from all lookup local
32766:  from all lookup main
32767:  from all lookup default
```

How is it possible that all packets are moved through the RPDB if there are no corresponding "to" rules? How is it that outgoing packets originating from the server don't get stuck? The answer is that every local packet - whether outgoing or addressed to *localhost* - still has a "from" or source IP address, regardless of its "to" or destination address. This fact allows all traffic to move through the rules because all traffic is coming "from" some address. The rules are checked in order from 0-32767. The kernel looks through the entire list to figure out which route is the most specific for the packet. The default rules simply pass all traffic through each table - local, main, default - in succession until a match is found or the packet fails to be matched to anything in any of these tables.

Routing Marked Packets with fwmark

The FireWall MARK (*fwmark*) can only be set by *iptables*, while both iptables and *ip rule* can read *fwmarks* (the *ip rule* system can only read its value and act on it as a *selector*).

Note *fwmark* converts decimal values to hexadecimal, though a query regarding its value may use either decimal or hexadecimal numbers. If referencing it in hexadecimal form, precede the value notation with an "x." For example, these two statements are identical:

```
ip rule add fwmark 32 lookup mytable
ip rule add fwmark x20 lookup mytable
```

Either statement will forward packets with an *fwmark* value of 32 to the routing table named "mytable."

iptables Explained

Recall the network diagram in Network Routes shows the process flow of a packet and when the packet may be acted upon by *iproute*, *iprule*, or *iptables*.

iptables is Ubuntu's built-in firewall. It uses its own set of rules to influence network traffic passing to, from, and through the server. Ubuntu 16.04.x uses *iptables* version 1.6. To verify your *iptables* version, execute the **iptables** command like so:

```
iptables --version
```

iptables may perform stateful packet inspection and mangle (manipulate) the packet to some extent, and has some control over the packet's path through the server. It differs from **ip route** and **ip rule** in that it cannot deliver a packet to a destination. Rather, *iptables* is only concerned with how the packet is handled inside of a server container, based upon the type of traffic.

Terminology Confusion

Some internet sources comingle the concepts of *iptables* and *ipchains*, inadvertently providing inaccurate explanations of how they function. Quite frankly, I can't blame other authors for misconstruing some of the concepts at work here. The architects of *netfilter* and the *ip* tools suite could have saved the public countless hours of confusion if they'd had the foresight to resist the temptation of recycling similar terms in the evolution of Linux networking. The relationship between *ipchains* and *iptables* is particularly confusing to many.

My intention is not to bad-mouth any other specific sources, but rather to draw attention to this fact. The distinction between *iptables* and *ipchains* is confusing, and this fact has been caused in no small part by the use of similar nomenclature to describe different processes and functions within *netfilter* and the Linux kernel. Generally speaking, Linux has a sordid history of old code never completely going away. Integrating new processes with legacy code always has been and continues to be the norm.

Chains and Tables

Iptables processes packets based on what are called *chains* - a carry-over from *ipchains* - that preceded *iptables*. *Chains* refer to *netfilter*'s hooks into the networking stack. *Tables* are rule-set containers inside *netfilter*.

Every packet is placed onto a *chain* based on the type of destination of the packet. From *iptables'* perspective, *chains* serve as conduits of process flow. Within that flow you will find hooks into various *tables*. Those *tables* are effectively repositories of rules. The net effect is *chains* appear to act as rule-sets, however that's not quite correct. In reality they are pointers to the rules stored in *tables*. When a packet traverses a chain, specific tables are referenced based on the particular chain. Within each referenced table, only rules that correlate to the active *chain* will be applied to the current packet.

This is the fundamental architecture behind *chains* and *tables*, and you should bear that in mind as you read through this section. These core concepts are not fully understood and incorrectly proselytized by many online references. If you can wrap your head around it, you will have no issues following the logic flow of rules as they are applied to packets and connections. The problem I find is many authors do not fully understand this concept and get confused about chains in particular. Chains don't control rules. They control the process, which tables are applied, and when.

Chains

Chains are all about process flow. A good way to think of *chains* is as collections of rules or pointers to rules. *Chains* determine the order in which sets of rules will be implemented. You cannot manipulate the processing order of the default chains, remove them, or change whether or not each chain is activated. What you can do is manipulate the rules in each chain (discussed in Tables).

Chains encapsulate the top-level process flow of packet filtering. *ipchains* preceded *iptables* and created the *chains* concept. It had just three chains, or processes: INPUT, OUTPUT, and FORWARD. They were named for the type of network traffic. At its most basic level of navigation, a network packet can only be travelling through a server along one of these three paths, and that is still true today. The INPUT chain relates to an incoming packet addressed to the current server. The OUTPUT chain is the opposite. It relates only to packets originating with the local host that are leaving the current sever. And FORWARD is reserved for packets that are neither and are simply traversing through the current server, where both source and destination are other servers.

Older iterations of *iptables* and *netfilter* used only the three aforementioned chains. The current version of *netfilter* has five built-in *chains*: *Input, Output*, *Forward, Pre-Routing, and Post-Routing*. These reflect the possible high level filtering routes of any packet traversing the server. *Iptables* also allows the creation of custom chains.

INPUT, OUTPUT, and FORWARD are *primary* chains, and reflect the three possible functional routes in which a packet may traverse the server: leaving (output), arriving (input), or traversing *through* the server (forward). Every packet goes through just ONE of those processes. The FORWARD chain is normally applicable only to firewalls and gateways (routers). You may generally ignore it if your server is not a router or firewall.

PREROUTING and POSTROUTING are *routing* chains, which work a little differently. All packets go through either PREROUTING (incoming) or POSTROUTING (outgoing), based solely on the direction of travel of the packet. The Pre and Post-Routing chains bookend the other chains, meaning they occur before or after the other chain. With the exception of the Forward chain, only one or the other will be called before or after the primary chains.

Here's a summary view of the five permanent chains:

- INPUT Local host is destination
- OUTPUT Local host is source
- FORWARD Host is neither source nor destination; applies to both INPUT and OUTPUT
- PREROUTING Executed before INPUT or FORWARD chains are processed
- POSTROUTING Executed after FORWARD or OUTPUT chains are processed

Default Chains

You may view the current status of **iptables** by executing this command:

```
iptables -L
```

By default this command will display the contents of the FILTER table in each chain. If you have not established any rules yet, your display will look like this:

```
Chain INPUT (policy ACCEPT)
```

```
target     prot opt source               destination

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
```

Roughly translated, each chain has a blanket ACCEPT policy, and all of the tables in each chain are empty. All traffic is allowed by default in each chain. That means all packets will be accepted or allowed to move through each chain.

Notice the PREROUTING and POSTROUTING chains do not appear when you list the current *iptables* rules. I haven't found a concrete explanation for why this is the case, but I suspect it is due to a combination of two factors: 1) A legacy characteristic of the migration to *iptables* from *ipchains*, because *ipchains* only used the three primary chains; and 2) by default the contents of the *Filter* table are displayed, which is only present in the three primary chains.

One of the major improvements from *ipchains* to *iptables* was limiting a packet to just a single primary chain, whereas the older *ipchains* method processed every packet through every primary chain, even if a chain did not apply to the packet. This architectural change alone resulted in significant speed improvements when *iptables* was introduced. Regardless, viewing the output shown above for the three primary chains is rarely useful since not all chains are displayed.

How To Create a Custom Chain

When is a *chain* not a *chain*? When you create one! Custom chains are possible, but they don't function in quite the same manner as the built-in chains. A custom chain acts like a programming sub-routine. When you call a custom chain, all of its rules will be acted upon, in priority order. When you create a new *chain*, you are creating a new rule container; a sub-set of rules. After you've created a custom *chain*, you populate it with rules. You may then reference the rules in that custom chain, and when that chain is called, those rules will be acted upon.

The logic flow confuses many people because it's different from the normal chains/tables process. Just like the default, static chains, when creating a new rule for your custom chain you must specify an associated table. However, custom chains may only be associated with the *Filter* table. Since the Filter table is the default condition when no table parameter is specified in a rule, you might as well leave it out of the commands for your custom chain.

Now, if you are astute, by now you should be wondering; How is it possible to access these new, custom chains? After all, there is no mechanism within the permanent chains to reference one another (and to do so would be contradictory to their function). The answer is by using a JUMP command that diverts the process flow to a non-terminating TARGET. This directs the current logic flow onto your custom chain.

Custom chains are very useful in complicated *iptables* configurations. They make it easier to organize and segment rules, grouping sets of oft repeated commands you want to replicate in different tables, different chains, or under different circumstances.

Custom Chain Examples

Take a look at an example of a custom chain.

Let's say you want to branch to a set of rules in your custom chain called "icmp-chain" when an ICMP packet is received on the INPUT chain.

1. First, create your new chain.

```
iptables -N icmp-chain
```

2. Append a command to the end of the INPUT chain that says to jump to the "icmp-chain" named chain when the packet protocol is ICMP. Note this will automatically be placed into the FILTER table. There's no choice on which *table* is utilized. However, you must indicate which primary or routing *chain* is associated with the redirection.

```
iptables -A INPUT -p icmp -JUMP icmp-chain
```

or

```
iptables -A INPUT -p icmp -j icmp-chain
```

3. Create at least one rule in your new chain. This one says to send an ICMP echo response

```
iptables -A icmp-chain -p icmp --icmp-type echo-reply -j ACCEPT
```

4. Don't forget to place a RETURN jump at the end of your chain, which will jump back to the next command in sequence after the command that called the custom chain. This is a good preventive practice. If a RETURN command does not exist and no actions are executed prior to reaching the end of the custom chain, *netfilter* will be unable to return back to where it started prior to jumping to the custom chain, the packet will be stuck and the connection will hang until it times out with a no-response error.

```
iptables -A icmp-chain -j RETURN
```

So, what's the point in custom chains?

The most common purpose of custom chains is to more clearly segment rules in a firewall. The process makes it easier to organize and call groups of rules that only pertain to specific circumstances. When those circumstances occur, the custom chain is called and its rules are operated upon.

Tables

Tables comprise groups of instructions defined in a particular order. *Tables* are called by *chains*. You cannot alter the built-in chains and tables or their order-of-execution.

You may think of *chains* as highways inside your server. They are able to direct traffic flow, but cannot modify the packets. The next layer down are *tables*: the filtering workhorse of *iptables*. They cannot direct traffic flow, but they may modify the packet. The end result is all filtering rules can be located in one place and called by the packet directional control processes (*chains*) as needed. A *table* simply contains a set of packet filtering instructions (but cannot alter which chain a packet traverses).

iptables has are five (5) static tables. They are: RAW, MANGLE, NAT, FILTER, and SECURITY. Of these, the Raw and Security tables are rarely used except within firewall servers. There is not a 1:1 correlation between *iptables* tables and *netfilter* chains.

Recall that *chains* represent conduits a packet travels through, depending on its source and destination. These default tables are called at various stages by each of the aforementioned chains. And just like the fact all packets don't traverse every chain, not all tables are called by every chain. This chart identifies which tables are called by each chain, and the order the tables are called:

- PREROUTING Raw, CONNTRACK, Mangle, NAT
- INPUT Mangle, Filter, Security, NAT
- OUTPUT Raw, CONNTRACK, Mangle, NAT, Filter, Security
- FORWARD Mangle, Filter, Security
- POSTROUTING Mangle, NAT

Here's another viewpoint demonstrating the tables associated with each default *iptables* chain:

- Raw PREROUTING, OUTPUT
- NAT PREROUTING, INPUT, OUTPUT, POSTROUTING
- Mangle PREROUTING, INPUT, OUTPUT, FORWARD, POSTROUTING
- Filter INPUT, OUTPUT, FORWARD
- Security INPUT, OUTPUT, FORWARD

I'm sure you noticed there are duplicate entries in these charts. Why is that? This is where the relationship between *chains* and *tables* begins to get a bit tricky. When constructing commands in *iptables*, they reside in tables. Yet commands are not executed until a chain is processed. Each chain calls the tables associated with the chain in sequential order, as expressed in the charts above. As each table is called, *netfilter* searches the table's contents for commands that match the current chain. Those matching commands are executed in order based on their priority in the table.

It's interesting to note some of the nuances of the default tables. For instance, only the MANGLE table is called by every chain. NAT is available in every chain except FORWARD, though only Source or Destination NAT are available in any given chain, depending on which chain is being processed. The SECURITY table always follows the FILTER table, and both are only available in the INPUT, OUTPUT, and FORWARD chains. This is interesting considering most rules on a firewall server normally reside in the Filter table. And as mentioned previously, firewalls are typically the only instance you're likely to see the SECURITY and RAW tables utilized.

Table Priority

Tables contain commands that influence the characteristics and/or the destination of a packet. In a firewall, your rules go in these tables. For the purpose of routing packets to a VPN, this is where rules are established to make that happen.

Let's examine the roles of each table:

- Raw — Specialized packet handling of unusual situations
- NAT — New connections go here after Raw table; modify based on IP address or port
- Mangle — Specialized packet alteration
- Filter — Traditional firewall rules; filter based on data type and source/destination
- Security — Specialized use for access control; gets called last if at all

RAW is rarely used and always called first. The *Raw* table allows a connection to circumvent other filtering. The *NAT* table can be used to modify IP address and port data in the packet. The *Mangle* and *Filter* tables perform the vast majority of packet manipulation. If you don't specify a table name in an **iptables** command, the *Filter* table will be used by default. The *Security* table is rarely relevant outside of firewall or router applications and it will be ignored in this guide as it is irrelevant for our purposes.

If no table is specified in a relevant **iptables** command, the *Filter* table is presumed by default. Not all tables are used in every process. When writing rules to influence VPN vs. non-VPN traffic, ensure you understand the order your rules will be processed, in order to facilitate the desired result. If a rule is post-ceded by a number, it indicates the order of operation (i.e. action #1 is performed before #2, etc.). Remember, if you don't specify a table, an instruction will be placed in the *filter* table by default.

Table Processing

Each table is called by one or more chains. Here are the chains associated with each built-in table.

- Raw — PREROUTING, OUTPUT
- NAT — PREROUTING, INPUT, OUTPUT, POSTROUTING
- Mangle — PREROUTING, INPUT, OUTPUT, FORWARD, POSTROUTING
- Filter — INPUT, OUTPUT, FORWARD
- Security — INPUT, OUTPUT, FORWARD

Studying the list above, you can see how the Mangle table is the most prevalent because it is called in every chain. Likewise, note the OUTPUT chain is the only one that calls every table.

High-Level Table Tips

Each of the five tables in *iptables* have unique qualities and deserve a bit of attention explaining why they exists and how they differ from one another.

Filter

The FILTER table is the default table loaded into the Linux kernel. If you do not apply any rules to a fresh Linux installation, the Filter table will still be present and populated with a very basic set of firewall rules. Its job is to filter packets with broad brush strokes.

You may think of the Filter table as a series of traffic lights that only have the red (stop) and green (go) colors. It contains only black-and-white rules that when applied, allow a packet to either continue or halt its journey. When one thinks generally of firewall rules, this is normally where those rules are stored and managed. A server and even a dedicated firewall can happily exist applying only the Filter table.

Mangle

In routing terms, "mangling" means the process of altering IP header packets before, during, or after routing. The *iptables* MANGLE table's job is to match packets against chains of rules and mark the packet so it can be routed appropriately. A common use of the Mangle table is to make adjustments to the priority of a packet based on *Quality of Service* (QoS)[5] and *Type of Service* (ToS)[6] rules.

The MANGLE table is unique in two regards: 1) All of its rules are analyzed; and 2) It is the only table that appears in all five chains.

All Rules Are Analyzed

Typically, when a rule inside a table is a match, that rule is executed and the packet then moves forward in the routing process. However, the Mangle table behaves differently from all other tables. The Mangle executes *every* matching rule against the current packet. This is one reason why applying packet filters in the Mangle table is strongly discouraged. Although it can be used for this purpose, the practice is dangerous and one must be extremely careful in rule application if one chooses to do so.

NAT

NAT is an acronym for *Network Address Translation*. The NAT table's job is - as it name implies - to translate network addresses. This includes the capability of masking outgoing packets so as to hide their origin within the local network. It also keeps track of packets that have been "masked" as such, sent out to an upstream router (e.g. somewhere on the Internet), and returned (requiring reverse translation so they can be sent to the originating server on the local area network).

Raw

The RAW table has a very narrowly defined role. Its sole purpose is to remove a packet from stateful inspection. This means the modified packet is not evaluated with regards to any other packet, and is evaluated discretely. This function may be applied to incoming or outgoing packets. This can be dangerous and use of the RAW table should be considered an advanced topic. Misuse can and will result in disrupted communications because the server will treat discrete packets as independent connections.

Security

The SECURITY table allows setting SELinux context to packets. It is perhaps the most esoteric of all the tables, the least likely to be utilized, and arguably the least understood. In a nutshell, it allows applying *Mandatory Access Control* (MAC) policies to a packet. However, it requires a pre-requisite Linux security module that is not present on most Linux distributions, making its use limited.

Astute observers may have noticed the Security table rarely appears on *iptables* networking diagrams. Why? For starters, it did not even exist until early 2008,[7] while *iproute2* has been around since 2000. Second, by default it is not applicable.

If your Linux distribution does not contain SELinux, iptables will ignore any attempts to affect the Security table.

The Security table's purpose is to set SELinux security context on packets pertaining to SELinux - an acronym for *Security-Enhanced Linux* (a Linux kernel security module). One of the hallmarks of Linux is the fact its kernel is built on a modular framework. A primary benefit of Linux is its ability to apply strict enforcement of access control policies and functions. SELinux applies a flexible framework of *Mandatory Access Control* (MAC) levers to the server's kernel. This allows capabilities such as controlling access to server resources by network connection and packet states.

SELinux is **not** embedded in Linux by default. It is enabled by default in some Linux distributions (such as Fedora and Red Hat), but not in others (such as Ubuntu). Alternatives also exist. For example, AppArmor, which as of this writing is compatible with the following Linux distributions: Arch Linux, CentOS, Debian, Gentoo, openSUSE, Pardus, PLD, and Ubuntu.

When present, the Security table is processed immediately after the Filter table.

Viewing Existing iptables Rules

Here's the syntax if you want to display information about your chains and tables via the command line. Remember, the default table (if you don't specify one) is *filter* and the default *chain* (if none is specified) is to display data from all chains.

```
iptables -t <table name> -L <chain>
```

For example,

```
iptables -t filter -L INPUT
```

Is the same as:

```
iptables -L INPUT
```

And will produce a result similar to this:

```
Chain INPUT (policy DROP)
target     prot opt source               destination
ACCEPT     all  --  anywhere             anywhere             ctstate ESTABLISHED
ACCEPT     all  --  anywhere             anywhere
DROP       all  --  anywhere             anywhere             ctstate INVALID
UDP        udp  --  anywhere             anywhere             ctstate NEW
```

What do those headings mean?

- **target** Specifies what should be done with the packet when a rule matches
- **prot** Protocol (e.g. TCP, UDP, ICMP, ALL)
- **opt** IP options; rarely used
- **source** Source IP address; or "anywhere" is not defined
- **destination** Destination IP address; or "anywhere" if not defined

You can also use the -S switch, which prints basically the same data as *-L* except it the actual rules are shown, just as if you'd type them into a file or command line (the **iptables** command prefix is omitted). For example,

```
iptables -S mychain
```

Would yield something like:

```
-N MYCHAIN
-A MYCHAIN -p tcp -m tcp --dport 22 -j ACCEPT
```

As you can see, the *-S* switch (*S*pecification) presents data as you'd type the commands, while the *-L* switch (*L*ist) displays a logical map of your ip rules.

If you viewed your chains after creating the example chain above in Custom Chain Examples, you would see the primary chains and your custom chains you've created (but you still wouldn't see the routing chains).

```
iptables -L
```

Would yield something like this:

```
Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination

Chain ICMP-CHAIN (0 reference)
target     prot opt source               destination
```

How to Delete Chains

Similar to displaying a list of rules, you may delete rules either by Rule Specification or by Chain and Number.

Deleting Rules by Chain and Number

Use the **--line-numbers** parameter to observe the line numbers of the rules in the appropriate chain.

```
iptables -L INPUT --line-numbers
```

Will yield output similar to this:

```
Chain INPUT (policy DROP)
num      target     prot opt source               destination
1        ACCEPT     all  --  anywhere             anywhere             ctstate ESTABLISHED
2        ACCEPT     all  --  anywhere             anywhere
3        DROP       all  --  anywhere             anywhere             ctstate INVALID
4        UDP        udp  --  anywhere             anywhere             ctstate NEW
```

Now, to delete a rule you would use the *-D* command. For example, to delete rule 3 from the INPUT table, you would type:

```
iptables -D INPUT 3
```

Deleting Rules by Specification

I personally find it much easier to delete rules by chain and rule number most of the time, but there is another, albeit cumbersome way to do it: contextually, by specifying a specific rule using identical rule text:

```
iptables -D <chain> [rule syntax]
```

Such as:

```
iptables -D INPUT -m conntrack --ctstate INVALID -j REJECT
```

Flushing Chains (Delete All Rules)

Sometimes you just want to delete all the rules in a chain and start over. There's an easy method of accomplishing this task; the Flush command (*-F*). Make sure you specify a chain, unless you wish to flush all the chains (see below).

```
iptables -F <chain>
```

Such as:

```
iptables -F INPUT
```

Flush All Chains

To flush all chains, type:

```
iptables -F
```

iptables Process Flow: Chains, Tables, and Rules

iptables often gets confusing because the same *tables* may be called from different *chains*. This begs the question: Do different chains call the same commands from tables with the same name? This is why chain and filter names are specified (implicitly or explicitly) in each command. The combination allows *netfilter* to skip over irrelevant code not tagged with the current chain or table name.

Recall that the network diagram in Network Routes shows the process flow of a packet and when the packet may be acted upon by *iproute*, *iprule*, or *iptables*. As previously described in Chains and Tables, by default all *iptables* commands are processed through five *chains* and five *tables*. Following the path of a network packet gets confusing because these concepts are intertwined with the Routing Policy DataBase (RPDB) process, and because it's so difficult to find literature that explains what these things are and how they are intertwined.

It's very important to follow the orderly logic of when each table's commands will be executed in each chain. While most documentation on this subject appears to reference tables in a way that gives the reader the impression tables are calling chains, I am convinced after studying *iptables* for some time that it is the other way around. Effectively, the chains dictate the path of a packet through *netfilter* and its handling by the kernel. The chains Input, Output, and Forward are the gatekeepers. No matter which table your rules are contained in, it is the chain the packet is associated with that determines which tables will be processed and when.

Another example supporting my point is the implementation of *iptables* rules doesn't steadfastly follow the concept where the table dictates which chains are called (and partly why I believe the reverse is actually true, as I've indicated above). For example, the **multiport** match extension protocol in *iptables* may be called during the PREROUTING chain. However, while the PREROUTING chain does not call the Filter table, the multiport command may also be used in the NAT table during processing of the PREROUTING chain.

I suspect much of the confusion - particularly of *iptables* neophytes - lies in the use of hidden or undeclared default values, such as the fact the filter table is the default location of new rules. Unless one knows to specify or declare certain characteristics of your rules and routes, *iptables* and even *iproute2* will often position things in places that aren't always logical to the average human being.

With regards to *iptables* in particular, if a packet is following a chain such that it never has a chance to encounter a table where some particular rule is located, the rule will obviously never be executed. The problem is this fact is easily obscured from casual observation. One must either learn the default locations and values of various aspects of *iptables* and *iproute*, or be diligent in always making specific declarations when writing commands.

To wit, let's re-cap the order of chain processing logic. In the "old days" of *ipchains*, only the primary chains existed, and the process was a bit easier to follow.

Primary Chain	Actual Chain Path
INPUT	PREROUTING -> INPUT -> Local Process
OUTPUT	Local Process -> OUTPUT -> POSTROUTING
FORWARD	PREROUTING -> FORWARD -> POSTROUTING

If you'd like a more visual description, review the charts in Network Routes and How Linux Routes Network Packets.

Here is the order of table processing within each *iptables* chain.

- PREROUTING Raw | CONNTRACK | Mangle | NAT (DNAT)
- INPUT Mangle | Filter | Security | NAT (SNAT)
- OUTPUT Raw | CONNTRACK | Mangle | NAT (DNAT) | Filter | Security
- FORWARD Mangle | Filter | Security
- POSTROUTING Mangle | NAT (SNAT)

You can think of *chains* as maps to and from a destination, and *tables* as waypoints along the route. When you add a new **iptables** command, a portion of the command line instructs *netfilter* which *chain* the command should be added to, and which *table* it belongs in within that *chain*. You can also think of it the other way around. Regardless, the end result is the same; when any particular command gets called, it is driven by the intersection of a *chain* and a *table*. No matter how you view their relationship, they are intertwined.

When you create a new command, you must decide at what point in the packet processing sequence it should be executed. "When" is a combination of which chain and which table within that chain. If you are familiar with software programming, you may liken this to the chain as a program and each table as a subroutine within the program. For example, if you wish to modify the source or destination address of a packet, you'd want to be sure you did that prior to a command that tells *netfilter* you're done processing the packet. If you reversed that order, the address modification would never happen. Of course, that sounds simple and logical, but in practice rules don't always get executed as you expect, so make sure you pay close attention to the order-of-execution of chains and the tables within each chain.

There's one more process in the mix you should be aware of. It's neither a chain nor a table, though you may have noticed it in the aforementioned charts and wondered what it was: it's called CONNTRACK.

CONNTRACK

CONNTRACK is an abbreviation for CONNection TRACKing. Recall the discussion of *fwmark* in IP Rule Syntax and Routing Marked Packets with fwmark. *FWMark* means *FireWall Mark* and relates to the action of "marking" or tagging packets. When a packet is marked, an internal value is assigned by *netfilter* creating an association with that particular packet. If the packet leaves the server and comes back, the mark will remain and *netfilter* will know it's seen the packet previously.

CONNTRACK tagging and inspection operands present an unusual scenario. They share all the *netfilter* hooks with chains except FORWARD, yet are processed independently of the chains and tables. The order-of-execution is always after the RAW table and before the MANGLE table in each chain respectively.

The *iptables* CONNTRACK function is discussed in greater detail under Match Extensions, part of the discussion on **iptables** command *parameters*.

NOTE: There is a *netfilter* process called *conntrack* and an iptables match extension function called **conntrack**. The conntrack module within *netfilter* and the **conntrack** match extension within *iptables* are independent of one another. However, their identical names can and do create considerable confusion. The key difference is the *iptables* **conntrack** reference is a match extension and read-only, while the CONNTRACK module in NetFilter is responsible for tracking and maintaining the current state of network connections.

Custom Chains

Custom chains must be created first. Then you may add rules to them. In order to utilize them, you must point to them from one of the five built-in chains. The chain commands above are used to create and delete custom

chains. Pointing to a custom chain is handled via *Parameters*. If you want to know more about Custom Chains and why you might want to create them, please review the section above, How To Create a Custom Chain.

Chain Policies

You may have noticed there is a command related to chains called *Policy*. What is a chain policy? A *policy* is simply a chain's default behavior. A chain's policy becomes relevant when the kernel reaches the end of a chain and there has been no matching rule that triggered an action or jump, or for whatever reason the packet was processed through a chain without making a determination of the packet's ultimate fate.

Chain policies can only be ACCEPT or REJECT. See Actions & Targets for more information.

iptables Command Line Syntax

There are a large number of commands and filters you may apply to your server's network rules using **iptables**. Some of these are found in extensions to *iptables*. Ubuntu 16.04 LTS includes many extensions installed by default. They allow more granular control over network packets under the context of *iptables*. Below, I've provided an explanation of a limited set of the available commands and filters. If you're hunkering for more info, I suggest you checkout the *iptables-extensions* Man Page and the official *netfilter* website. The Unix and Linux Forums website also maintains an excellent man page for iptables that includes a lot of helpful details on the command set.

The **iptables** command line syntax is:

iptables {table} [COMMAND] [CHAIN] {parameter} {extension} [ACTION] {target}

Important syntax notes for **iptables** commands:

- May be applied to packets or connections
- COMMAND, CHAIN, and ACTION are required values
- *Commands*
  - Always UPPERcase
  - Only one command instruction per line
  - Define actions to be performed on a packet matching a specified, corresponding rule
- *Table*
  - Use *-t* prefix (for example, "iptables -t INPUT"
  - If no table name is specified, the FILTER table is presumed

Let's look at an example and break it down.

iptables -A INPUT -i eth1 --src-range 8.8.8.0/24 -j ACCEPT

In the example above, we have a command with these values:

iptables {COMMAND} {CHAIN} {PARAMETER} {EXTENSION} {ACTION}

Where are *{TABLE}* and *{TARGET}*? The *chain* must be specified, but the *table* is optional. When a table is not specified, by default the rule is inserted into the FILTER table. No *target* is specified in the example above because there is no target. So, in the case of this example, the rule may be roughly interpreted as,

"Append to the end of the INPUT chain, accept the packet if all of the following conditions are true: the Interface is eth1 AND the Source IP Address is between 8.8.8.0 and 8.8.8.255"

Reading each segment in the line: **-A** is the command; **INPUT** is the chain being acted upn; **-i** indicates the next value will be the interface (**eth1**); **--src-range** narrows the filter to a specific range of IP addresses (**8.8.8.0/24** or 8.8.8.0-8.8.8.255).

The ACCEPT action tells *netfilter* to accept the packet and stop processing the packet in the current chain. Matching a rule with an ACCEPT target means the processing of this packet is finished for this chain. The packet will then proceed through the remainder of the network packet filtering process. If there are other chains after this one, they will still need to accept the packet in order for it to continue moving on.

Applying iptables Commands to Chains

Aside from applying *iptables* commands to IP packets, you may also create rules that control the chains themselves. This is particularly useful when you have created custom chains.

Here's a list of the important commands that affect chains:

- -A *chain rule* Append (append rule to the end of the selected chain)
- -D *chain {rule}* Delete rule in specified chain using rule specification
- -D *chain {rule #}* Delete rule *x* in specified chain where *x* is rule number
- -E *old-name new-name* Exchange (replace) old chain name with new chain name
- -F *chain* Flush selected chain or table (delete all rules); all chains flushed if none specified
- -I *chain {priority}* Insert new rule into a chain; 1 [default] = insert in front of any other existing rules)
- -L *chain* List all rules in specified chain[8]
- -N *chain name* Create a new chain[9]
- -P *chain target* Policy (must be ACCEPT or DROP)
- -S *chain* List rule specifications in specified chain
- -X *chain* Delete custom chain (by name)

Next, I'll group them for you to make it easier to figure out which command you need for whatever you're trying to accomplish.

Actions Upon Chains

- -F *{chain}* Flush selected chain or table (delete all rules); all chains flushed if none specified
- -N *chain name* Create a new chain
- -P *chain target* Policy (must be ACCEPT or DROP)
- -X *chain name* Delete custom chain

Actions Upon Rules

- -A *chain rule* Append (append rule to the end of the selected chain)
- -D *chain {rule}* Delete rule in specified chain using rule specification
- -D *chain {rule #}* Delete rule *x* in specified chain where *x* is rule number
- -I *{chain} {priority}* Insert new rule into a chain; 1 [default] = insert in front of any other existing rules)

Parameters

Parameters are optional filters that further narrow the scope of commands. For example, the "-p" or "-protocol" parameter defines a protocol that restricts the action of the command line to 1) a specific chain; and 2) a specific parameter within the chain. So, in an example with "... INPUT -p tcp..." for instance, the action and target would apply only to the INPUT chain and within that chain, only to TCP/IP traffic. Parameters are always expressed in lower case.

- -d *addr /mask* Destination address; may be network name or IP, hostname, external IP

- -i *interface name* Name of interface In (receiving packet); only INPUT, FORWARD, PREROUTING
- -m Match; tries to match conditions in order (all criteria must be met)
- -o *interface name* Interface out (sending packets); only FORWARD, OUTPUT, POSTROUTING chains
- -p Protocol of a packet (TCP, UDP, ICMP, or ALL)[10]
- -s *addr /mask* Source address; internal or external network name/IP, hostname

iptables Extensions

There are dozens of parameter filters you may implement in *iptables*. Due to the manner in which *iptables* man pages tend to order extension explanations, digesting their syntax can be quite confusing. You may think of *parameters* as coarse filters for *iptables*, and *iptables extensions* as fine filters for more granular control over packets and connections.

For example, *-protocol* (shortcut *-p*) is a parameter that can be used to restrict an **iptables** rule to a particular network protocol (e.g. *-p tcp* or *-protocol udp*). However, it's also possible to extend that capability and create a more granular filter, such as restricting a rule to a particular networking protocol (e.g. TCP) and further narrowing it down to a defined source or destination port range. While the former concept is a capability built-in to *iptables*, the latter concept requires the use of *iptables extensions*.

Extensions are grouped into various modules. The modules are not called in the command line. Rather, they contain groups or sets of extensions, and those extensions are the *parameters* called in *iptables* rules. It gets confusing when attempting to comprehend whether you need to use the extension name itself and/or a particular extension name sub-set. From an implementation perspective, it boils down to understanding which parameters to use to accomplish your goals. We'll focus primarily on two extension sub-sets: *Match* and *Target* extensions. These are the most commonly used *iptables* extensions, regardless of the purpose of the server.

Match Extensions

One of the most powerful tools in **iptables** are *match extensions*. They are loaded via the *-m* parameter.

The syntax format of all *iptables* extensions is:

-{parameter} --{extension} {option} {argument}

So, in the case of *match* extensions, this portion of the *iptables* command line looks like this:

-match --{extension} {option} {argument}

Or the shorthand version:

-m --{extension} {option} {argument}

Following the *-m* parameter, *match* uses options and arguments to filter traffic. Multiple match options and argument variants are permitted on the same line. For instance, matching a connection state and a protocol at the same time. Just a portion of the complete set of match extensions are relevant for a media server.

Let's take a look at the relevant *match* extensions and what characteristics they match. As previously mentioned, the *match* extensions are grouped into modules. Within each module is a set of extensions and/or options. The relevant *match* extension modules are:

- connmark *netfilter* marked fields associated with a connection
- conntrack Connection tracking
- iprange Range of IP addresses
- mac Source MAC ID
- mark *netfilter* marked fields associated with a packet
- multiport Set of source or destination ports
- owner Outgoing packets based on *userid* or *groupid*

Bear in mind multiple match extensions may be grouped together on the same line. Needs will vary, but these are the extensions most likely to be useful, grouped by function. If you can't find a function to suit your needs here, for Ubuntu check the official man page here. Or for non-Ubuntu operating systems, check the iptables man page for Linux, or for either simply request the man page via your terminal:

```
man iptables
```

Actions and Targets

An action is a parameter applied at the end of a line, just before the target. The most common actions are *-goto* and *-jump*, which are similar but are distinctly different in their behavior.

Goto

The *-goto* or *-g* action instructs *iptables* to 'goto' to a custom chain name specified after the *-g*. When *iptables* finds a matching rule containing a *-goto* command, it stops processing the current chain and jumps or goes to the specified *target custom chain name*.

-g my_custom_chain_name
-goto my_custom_chain_name

Jump

The *-jump* or *-j* action instructs *iptables* to 'jump' to the target identified after the *-j*. When *iptables* finds a matching rule containing a *-jump* command, it stops processing the current chain and performs the specified *target action*.

The *target* is the last item in an **iptables** command line. The most common are the built-in targets mentioned in the previous section: ACCEPT, DROP, and REJECT. *Target* indicates the target process or what the desired behavior is if all conditions of the line are met.

-j ACCEPT
-j DROP
-j REJECT

ACCEPT ends the processing of the current packet within the current chain. From there it goes on to the next chain in the order previously discussed. If the packet is currently in the POSTROUTING chain, when it reaches an ACCEPT target, all mangling of the packet ends and it is sent out of the server.

The difference between DROP and REJECT is DROP simply drops the connection, while REJECT drops the connection and returns a message to the requesting process, advising it the packet was rejected. DROP provides no notification.

Match Extensions

Match extensions identify packets that match a specified criteria. This document focuses only on the most frequently used match extension modules, which are:

- comment
- connmark
- conntrack
- iprange
- mac
- mark
- multiport
- owner
- state [deprecated]

These modules are explained below.

connmark

Part of the ConnTrack module (CONNection MARKing), this has to be the worst-named extension. Why do I say that? Because there are **two** ConnMark extensions. One is a match extension, which is *connmark*. The other is a target extension, which is *CONNMARK*. Confused yet?

connmark matches connections marked with the *netfilter* mark field (see MARK target extension). Use it to check for the presence of a specific mark value associated with a connection.

-m connmark --mark <value>

```
iptables {table} {COMMAND} {CHAIN} {parameter} -m connmark --mark 10 {ACTION} {target}
```

conntrack

Conntrack allows access to the connection tracking state for the current packet/connection. It can be used to match connections based on state, protocol, source/destination originating IP addresses, TTL (Time To Live), reply source/destination IP addresses, and flow direction (originating or reply). Most of those features are not explained here as they can be managed with other commands. The most important function of *conntrack* is its ability to act based on connection state.

You may query the state of the connection using the extension *--ctstate*.

-m conntrack --ctstate <statelist>

--ctstate is unique compared with the other match extensions. It branches based on a connection's state. The *statelist* includes the following valid states. You may use more than one as the argument:

- NEW First packet of a connection

- ESTABLISHED Packet is part of a known 2-way connection
- RELATED Packet is starting new connection, but associated with an existing connection
- INVALID The packet has no known associated connection
- UNTRACKED Packet is exempt from tracking per NOTRACK command in RAW table
- SNAT Virtual state. Matches if source IP address has been altered
- DNAT Virtual state. Matches if destination IP address has been altered

Example syntax:

```
iptables ... -m conntrack --ctstate RELATED {ACTION} {target}
```

NOTE: There is a *netfilter* process called *conntrack*, which is responsible for tracking and maintaining the current state of network connections. The conntrack module within *netfilter* and the **conntrack** match extension within *iptables* are independent of one another. However, their identical names can and do create confusion. The key difference is the *iptables* **conntrack** reference is a match extension and read-only.

iprange

IPrange allows you to create a match rule based on a specified IPv4 address or range of addresses. You must specify if the address(es) to be matched are source or destination addresses. The format is:

-m iprange --src-range from{-to}

-m iprange --dst-range from{-to}

mac

A Media Access Control or MAC address is a unique identifier assigned to a network interface controller. MAC addresses are tied to a specific device, rather than a specific IP address.

-m mac --mac-source {XX:XX:XX:XX:XX:XX}

The MAC address match must be called with -m mac. The MAC source is specified with the **--mac-source** argument, after the MAC module has been invoked with **-m mac**. If you forget to include the **-m mac** prefix in the command, it won't work.

mark

Matches packets with a previously associated *mark* value. A *mark* may be assigned via any of three methods:

1. The current connection was previously marked with the CONNMARK target extension
2. The current packet was previously marked via the MARK target extension
3. Via a rule set in *ip rule* that sets a corresponding *fwmark* (FireWall mark)

The syntax of the *mark* match extension is:

-m mark --mark {value}

multiport

Muliport allows matching based on source, destination, or both direction of traffic flow (source/destination) and a single port, group of ports, or range of ports. The maximum number of ports specified as a command line argument is 15. Note that a range (e.g. x-y) counts as two (2) ports out of the maximum of 15.

-m multiport {--source-ports | --destination-ports | --ports {port,port,port:port}

Like all match extensions, *multiport* may be combined on the same line with other match extensions. For example, you could match the TCP extension with the multiport extension. Here is an example that matches a condition where the protocol is TCP and the port the packet is coming either to or from is either 80 (HTTP) or 443 (HTTPS):

```
iptables ... -p tcp -m multiport --ports 80,443 {ACTION} {target}
```

Note there is one exception in syntax that should be noted. *Multiport* may not be combined with standalone --source-port or --destination-port matches when the multiport command corresponds with the same action as --source-port or --destination-port. If you do combine them in the same command, the first related action will be executed and subsequent actions affecting the same module will be ignored.

owner

The Owner module matches packets with the local owner (of the packet) that corresponds to the match criteria. This can be a single UsernameID, single UserGroupID, a range of UserIDs, or a range of GroupIDs. Ranges may only be specified if numeric arguments are used (i.e. userid or groupid).

Owner is only applicable in mangle table of the OUTPUT and POSTROUTING chains. If attempts are made to apply it in any other chains or tables it will be ignored. Possible applications are:

--uid-owner {username}
--uid-owner {userid | userid-userid}
--gid-owner {groupname}
--gid-owner {groupid | groupid-groupid}

Examples in context with *-m* switch:

-m owner --uid-owner username

-m owner --uid-owner userid1

-m owner --uid-owner userid1-userid2

-m owner --gid-owner groupname

-m owner --gid-owner groupid

-m owner --gid-owner groupid1-groupid2

state (Deprecated)

Although the *state* match exists, this extension should not be used as it was deprecated in *iptables* version 1.4.16 (Edwards & Engelhardt, 2013). It continues to function but is aliased to the *conntrack* module (**-m**

conntrack), meaning ConnTrack functions are what is called by *state*, and therefore ConnTrack's functions are the ones you should be using. While currently functional for backwards compatibility, the *state* match extension should be avoided and *conntrack* should be used instead.

Since you may see this match parameter represented in documentation, you should be aware of its (now deprecated) syntax:

```
iptables ... -m state --state <statelist> ...
```

The *statelist* utilized by *state* is the same as ConnTrack's.

--source-port (--sport)

--source-port (or **--sport**) allows matching based on a single, group, or range of source ports. The maximum number of ports specified as a command line argument is 15. A range (e.g. x-y) counts as two (2) ports out of the maximum of 15.

-p {protocol} --source-port {port,port,port:port}

Here is an example that matches source ports 52, 53, 80, and 443:

```
iptables ... -p tcp --source-ports 52-53,80,443 {ACTION} {target}
```

Note --*destination-port* and --*source-port* are protocol specific. The protocol must be specified on the command line, and must precede the -port command. The protocol can only be one of TCP, UDP, or ICMP.

Target Extensions

Target extensions affect the final disposition or destination of a packet or a connection. This could mean halting a particular packet or connection, tagging it, or redirecting its destination. A *target* extension need not necessarily end a packet's or a connection's processing tree, though some do.

Target extensions are invoked via the jump action (-j | -jump).

There are three (3) target types built-in to the core *iptables* code:

- ACCEPT
- DROP
- REJECT

A *target module* is one that performs an action against the current packet or connection. The following additional target modules are added by the target extension module:

- CONNMARK — Matches packets based on a *mark* value
- CONNSECMARK — Assign a security mark to a connection
- CT — Disables tracking for current packet when used with *--notrack* option
- DNAT — Alters destination address of incoming packets
- MARK — Assigns a *mark* value to a packet
- MASQUERADE — Conceal internal IP address when sending packets outside the LAN
- NOTRACK — Do not track the state of the current packet
- REJECT — Drop packet and return error
- SECMARK — Assign a security mark to a packet
- SNAT — Change source IP address to specified value
- --destination-port — Protocol specific matching by destination port number(s) [TCP | UDP | ICMP]
- --source-port — Protocol specific matching by source port number(s) [TCP | UDP | ICMP]

CONNMARK

Set the *netfilter* mark associated with a connection. The mark must be an integer (32-bits).

Syntax:

- --restore-mark — Copy the *connection mark* to the *packet mark*
- --save-mark — Copy the *packet mark* to the *connection mark*

Example:

```
iptables -A POSTROUTING -t mangle -j CONNMARK --restore-mark
```

CONNMARK target commands may be used in any chain and in any table. However, it is recommended to use them only in the *mangle* table (Andreasson, 2006; p. 215). Unpredictable results may occur used in other tables.

Note the important distinction between CONNMARK (connection) and MARK (packet).

CONNSECMARK

Assigns a security mark to a connection.

SELinux is beyond the scope of this document. However, due to legacy code in *iptables*, a small number of functions in SELinux have comparable functions in iptables that may be accessed even when SELinux is not present. This section briefly describes one of those functions, for the sake of completeness.

This is an oddball. Ideally, it belongs only in the security table of the INPUT, FORWARD, and OUTPUT chains. However, in the event you're working with a host not running a SELinux security module, there is a legacy entry point of sorts. This extension can be utilized via the *mangle* table of the PREROUTING chain.

Syntax:

- --restore   If no SECMARK exists on the packet, copy the *connection security context mark* to the *packet*
- --save   If a security context mark exists on the packet and no security context mark exists on the connection, copy the *packet security mark* to the *connection*

Example:

```
iptables -A POSTROUTING -t mangle -p tcp -dport 80 -j CONNSECMARK --restore
```

CT

"CT" is an acronym for "*Connection Target*." Exclusive to the RAW table, the CT target extension allows one to set parameters for a packet OR its associated connection. I only mention CT in this document as an alternative to removing connection tracking in the RAW table. CT basically creates and applies a connection tracking template to an incoming packet, which is then overlaid onto the *conntrack* module in *netfilter*. This is why it's only relevant in the RAW table.

Syntax:

- --notrack   Disable connection tracking for the packet

CT has other capabilities, but they are quite advanced and well beyond the scope of this document.

DNAT = Destination Network Address Translation

DNAT is acronym for *Destination Network Address Translation*. DNAT is the opposite of SNAT. It determines where a packet should go on the *local* server. Applied only during the PREROUTING chain, when the packet has just entered the server. This ensures future traffic over the same connection has the DNAT change applied to it.

DNAT rewrites the *destination address* of the packet.

DNAT translates an incoming public IP address into a specific private (LAN side) IP address. A typical use case would be when you always want traffic directed to a certain public IP address to be routed to a certain internal server.

Port Forwarding[11] uses DNAT in combination with port filtering. DNAT may also be used to force protocol filtering, which while very similar to port forwarding, causes incoming network traffic to be redirected based on protocol (e.g. UDP). This results in a wider scope of packet redirection.

Note that port forwards may include a range of port numbers. Furthermore, a neat trick is it may be used to change destination port numbers without affecting the IP address. For example, this command funnels all incoming TCP/IP traffic on interface eth0 to be sent to port 80, without changing the destination IP address:

```
iptables -t nat -A PREROUTING -p tcp -o eth0 -j DNAT --to-destination :80
```

Dynamic DNAT

There is a variation of DNAT sometimes referred to as *Dynamic* DNAT or *Dynamic Mode* DNAT. Dynamic DNAT is simply a list of range-bound DNAT addresses, any of which may be assigned to the current connection. *Iptables* will select the least frequently used address (a form of load-balancing). For example:

```
iptables -t nat -A PREROUTING -j DNAT --to-destination 192.168.0.21-192.168.0.25
```

MASQUERADE

MASQUERADE is an alternative to SNAT. It is applicable to outbound connections and works almost exactly like SNAT, except it cannot force a public IP address. Instead, it passes the public IP address of the router, thus masquerading or disguising the internal IP address of the originating server. MASQUERADE requires Connection Tracking to be in use on the current connection (e.g. it won't work if state has been de-activated via a command in the RAW table).

MASQUERADE may only be used with one of the following protocols: TCP, UDP, DCCP, or SCTP.

MASQUERADE may only be applied in the nat table of the POSTROUTING chain.

The following options may be applied:

- --to-ports Specifies a range of source ports to use
- --random Randomizes source port mapping

MARK

A numeric field (32-bit integer) used to mark (flag) a packet by associating a value with it. This is the same thing as fwmark in *ip rules*. The MARK command in *iptables* **sets** a packet mark, while packet marks may be read and acted upon by *iptables* and/or *ip rules* (per **fwmark**).

The MARK command can only be used in the MANGLE table. It is normally applied in the PREROUTING chain, but may be applied in the INPUT or OUTPUT chains as well. To be used in conjunction with a corresponding **fwmark** rule in *ip rules*, the mark command must be in the PREROUTING chain (mangle table).

The syntax is:

- --set-mark

Example:

```
... -j mark --set-mark 32
```

Note the important distinctions between CONNMARK (connection), MARK (packet), and fwmark (*ip rules*).

NOTRACK

Disables connection tracking for all packets matching the rule. Equivalent to using *CT --notrack*. As with CT, this function can only be used in the RAW table. It may be applied in the PREROUTING or OUTPUT chains.

REJECT

This is effectively the same as executing a DROP command (via "jump to," i.e. *-j DROP*). The difference is the REJECT command will return an error packet to the server on the other end of the connection, informing it the packet was rejected.

SECMARK

Assigns a security mark to the current packet.

SELinux is beyond the scope of this document. However, due to legacy code in *iptables*, a small number of functions in SELinux have comparable functions in iptables that may be accessed even when SELinux is not present. This section briefly describes one of those functions, for the sake of completeness.

This is an oddball. Ideally, it belongs only in the security table of the INPUT, FORWARD, and OUTPUT chains. However, in the event you're working with a host not running a SELinux security module, there is a legacy entry point of sorts. This extension can be utilized via the *mangle* table of the PREROUTING chain.

Syntax:

-A {chain} -t {table} {parameters} -j SECMARK {security context}

Example:

```
iptables -A POSTROUTING -t mangle -p tcp -dport 80 -j SECMARK --selctx httpcontext
```

SNAT = Source Network Address Translation

SNAT is acronym for *Source Network Address Translation*. SNAT is the opposite of DNAT. Applied only during the POSTROUTING chain, just before the packet leaves the server. This ensures future traffic over the same connection has the SNAT change applied to it.

SNAT rewrites the *source* IPv4 address of the packet and is used in conjunction with the *-o* parameter (e.g. "-o eth0") to modify an *outgoing* packet on the given interface.

For example, this command funnels all outgoing TCP/IP traffic on interface eth0 to be sent to port 80, without changing the source IP address:

```
iptables -t nat -A POSTROUTING -p tcp -o eth0 -j SNAT --to-source :80
```

Dynamic SNAT

There is a variation of SNAT sometimes referred to as *Dynamic* SNAT or *Dynamic Mode* SNAT. Dynamic SNAT is simply a list of range-bound SNAT addresses, any of which may be assigned to the current connection. *Iptables* will select the least frequently used address (a form of load-balancing). For example:

```
iptables -t nat -A POSTROUTING -o eth0 -j SNAT --to-source 192.168.0.1-192.168.0.10
```

Extensions and Chains: Where and When to Apply *iptables* Rules

Most *iptables* extensions cannot be applied to every chain. For some extensions, it's very difficult to track down information on which extensions work in which chains and tables. From what I have ascertained through many hours of research, at least half of them are completely void of information on their chain and table limitations. That said, there are typical use scenarios. Consider what you are trying to accomplish before creating your *iptables* commands, and whether or not it makes sense to apply them to a particular chain or table.

Match Extension Chains and Tables

Here is a list of each match extension defined above and the chains in which they typically belong:

- connmark PREROUTING, POSTROUTING
- conntrack INPUT, PREROUTING
- iprange PREROUTING, INPUT, FORWARD, OUTPUT, POSTROUTING
- mac PREROUTING, INPUT, FORWARD
- mark INPUT, PREROUTING, POSTROUTING
- multiport INPUT, FORWARD, OUTPUT, PREROUTING
- owner OUTPUT, POSTROUTING

And here is the list of known tables where you may apply each of those match extensions:

- connmark mangle, filter
- conntrack filter
- iprange filter
- mac filter
- mark mangle
- multiport nat, filter
- owner filter

Target Extension Chains and Tables

Here is a list of each target extension defined above and the chains in which they are typically used.

- CONNMARK PREROUTING, INPUT, FORWARD, OUTPUT, POSTROUTING
- DNAT PREROUTING, OUTPUT
- MARK PREROUTING, INPUT, OUTPUT
- MASQUERADE POSTROUTING
- NOTRACK PREROUTING, OUTPUT
- REJECT INPUT, FORWARD, OUTPUT
- SNAT POSTROUTING

And here are their corresponding tables:

- CONNMARK mangle
- DNAT nat
- MARK mangle
- MASQUERADE nat
- NOTRACK raw

- REJECT nat, filter
- SNAT nat

Inverse Operand

It is possible to precede a match parameter with the inverse operand (" ! "). This has the effect of a "NOT" operation. The **!** symbol precedes the -- in front of the extension option. For example, let's take a look at a portion of a command line filtering based on an IPv4 range.

```
... -m iprange --src-range 192.168.1.1-192.168.1.255
```

Normally, this match extension would limit the *iptables* rule to the range of IPv4 source addresses between 192.168.1.1 and 192.168.1.255. If you placed an inverse operand character in front of the *--src-range* option, you would instruct *iptables* to do the opposite.

```
... -m iprange ! --src-range 192.168.1.1-192.168.1.255
```

The command above would instruct *iptables* to apply the rule to any IPv4 source address NOT between 192.168.1.1 and 192.168.1.255. The opposite effect of the original command option.

Breaking that down, we have:

- -m is the iptables extension being called (in this case, "match")
- *iprange* is the name of the filter
- *--src-range* (source range) is the option being called within the iprange filter
- 192.168.1.1-192.168.1.255 are the affected range of IPv4 addresses
- Alternatively, the use of *! --src-range* (source range) instructs *iptables* to action all source IPv4 ranges except for 192.168.1.1-192.168.1.255.

Noteworthy Variable Syntax

You may have a need to know about a few other variables, depending on the commands you choose to implement.

- LO Loopback ; applicable to INPUT and OUTPUT chains
- /mask Genmask; post-cedes IPv4 format /x.x.x.x or /xx (e.g. /255.255.255.0 or /24)

Connection Tracking

As explained under CONNTRACK, connection tracking is an extension to *iptables* (also known as modules) that allows monitoring the status of a network connection. All traffic coming into or going out of your server is transmitted as network packets, and each of those packets is part of a connection. New connections are those where the packet is the very first packet. It could be an outgoing packet that originated from your server or an incoming packet arriving from another host. The Connection Tracking or ConnTrack module is unusual in that it adds both *match* and *target* extensions to *iptables*.

Protocols

Protocols operate in a similar manner to *Match Extensions*. Protocols filter the current packet based on the current communications protocol. There are only a few options, and for our purposes we will only focus on the two most common: TCP and UDP. Protocol filtering may be combined with match extensions (-*m*) on the same line to create more sophisticated filters on a single command line.

TCP

Match extensions that filter based on port numbers must be used in conjunction with a protocol (e.g. *-p tcp* or *-p udp*). Port filtering may consist of a single port, group of ports, range of ports, or a group of port ranges. The maximum number of ports specified as a command line argument is 15. A range (e.g. x-y) counts as two ports (out of the maximum of 15).

Here are some syntax examples of the **tcp** *protocol* used with various port number filters.

-p tcp --source-port {port,port,port:port}

-p tcp --sport {port,port,port:port}

-p tcp --destination-port {port,port,port:port}

-p tcp --dport {port,port,port:port}

-p tcp --dport 0:9999,12000:15000,34999

Note there is no module prefix between the -m parameter and the match extension syntax.

UDP

The UDP match extension follows the same format as the TCP extension.

Port filtering may consist of a single port, group of ports, range of ports, or a group of port ranges. The maximum number of ports specified as a command line argument is 15. A range (e.g. x-y) counts as two ports (out of the maximum of 15).

Here are some syntax examples of the **tcp** *protocol* used with various port number filters.

-p udp --source-port {port,port,port:port}

-p udp --sport {port,port,port:port}

-p udp --sport 23,47:125,57:123,449

-p udp --destination-port {port,port,port:port}

-p udp --dport {port,port,port:port}

Note there is no module prefix between the -m parameter and the match extension syntax.

Testing Your Rules

It may not be obvious, but you need to test your rules - both routing rules and *iptables* rules - before you cement them in place. One of the great things about the RPDB and *iptables* is they are very forgiving of catastrophic screw-ups. Simply force a server reboot, your rules are emptied, and you may start over. So, before you make them permanent (automatically load on boot-up), it's critical to thoroughly test them, lest you lock yourself out of your server. I can't stress this enough with a headless server.

I've found the best way to do this is to create custom scripting files that populate the rules in your RPDB and *iptables*. You can certainly also populate your route tables with this method as well, and I'll show you how to do that as well.

Listing Rules by Number

Here's how to list all the rules in all the chains with line numbers so you can identify each rule's number. This can be very useful when troubleshooting or to identify a rule number for deletion.

iptables -L -v -n --line-numbers

You may specify a particular table if you know which you want to focus on. Simply include the table name after the *-L* command.

Create Your Shell Scripts

You should create two scripts. One for *iptables* and one for your RPDB rules and/or routes. The reason why is because the implementation process that will automatically load your rules needs to be segmented.

iptables Shell Script

Open your favorite text editor and begin a new file. Populate the file with your *iptables* entries. This is a very basic example. Your needs will vary, and this guide's purpose is simply to provide an overview of Linux networking in Ubuntu and various processes. You must figure out what *iptables* rules you need and modify the sample script below to meet your needs.

```
nano /etc/iptables-script.sh
```

Here's a very basic sample script. Notice at the end of the file it calls the next file, which will update your RPDB rules and routes.

```
#! /bin/bash
# iptables sample test script

# flush iptables rules
iptables -F -t nat
iptables -F -t mangle

# OUTPUT chain
iptables -t mangle -A OUTPUT -j CONNMARK --restore-mark

# launch RPDB script
/etc/routing-script.sh
```

```
exit 0
```

Save the file.

Make the script file executable.

```
chmod +x /etc/iptables-script.sh
chmod 755 /etc/iptables-script.sh
```

Routing and RPDB Rules Shell Script

You'll now create an independent script to manage changes to the RPDB routes and rules. If you plan on using custom routing tables, make sure you've already created them in the master routing table. Instructions on how to do this are found in The Master Routing Table.

Begin a new script file. Remember, this will be called by the first file. Ensure the filename reference at the end of your first script matches this filename.

```
nano /etc/routing-script.sh
```

Here's a very basic sample script that will update your RPDB rules and routes. Remember, this is just a sample file. Do not use this verbatim!

```
#! /bin/bash
# RPDB sample test script

# presume there is an existing table called custom-table-name
# you would have created this table previously in the table names file

# sample ip rule
ip rule add from 192.168.1.1 lookup custom-table-name

# sample ip route
ip route add default via 192.168.1.1 dev eth0 table custom-table-name

# reload cache to load new rules and routes
ip route flush cache

exit 0
```

Save the file.

Make the script file executable.

```
chmod +x /etc/routing-script.sh
chmod 755 /etc/routing-script.sh
```

To test your changes, simply execute the first script.

```
./etc/iptables-script.sh
```

Helpful Tips

Here are a few helpful tips related to rule and route testing.

Don't Forget to Flush!

If you make changes to *ip rules, ip routes, or iptables* and wish to utilize them prior to the next system reboot, don't forget to flush the cache before you change *iptables* and after you change *ip routes*. Flushing the cache forces the kernel to reload the *iptables* and routing databases.

Do NOT Flush ip rules

It's important to flush the cache for routes and tables, but do NOT do that for ip rules. It will break your server's connectivity.

mark vs. fwmark: What's the Difference?

Are you banging your head into a wall trying to understand the relationship between "mark," "fwmark," and "firewall mark?" Well, the good (and simple) news is *they are all the same thing*. The difference is *fwmark* is the reference name used by *ip rules*, and mark is the reference name used by *iptables*. All three of these terms refer to "marks" applied by the CONNMARK and CONNTRACK *netfilter* modules.

Persisting Routes, Rules, and Tables Across Reboots

Routing Policy DataBase (RPDB) rules and *iptables* commands need to be loaded into the kernel's memory when the server starts up. The *master routing table* is the only portion of the RPDB that will automatically repopulate after a server restart. The remainder of the RPDB and *iptables* commands must be reinstated after every reboot. There are several methods of doing this, but it can be a little tricky as of Ubuntu 16.04 due to significant infrastructure changes from 14.04.

SystemD and the Start-up Process

Now that you've setup and tested your scripts, it's time to configure your server to automatically execute them when it boots. This will cause your rules and routes to be established every time the server restarts, which is necessary because otherwise they won't be present.

To make this happen, you need to create a hook that calls your script file during the server's startup process.

1. If you just rebooted after testing your scripts, make sure you login as root.

```
sudo -i
```

2. Create a **.service** file. Let's name it *rules.service*.

```
nano /etc/systemd/system/rules.service
```

3. Add the following lines. Note the filename matching your first test filename.

```
[Unit]
```

```
Description=rules service

[Service]
ExecStart=/etc/iptables-script.sh

[Install]
WantedBy=multi-user.target
```

4. The script files must be executable. Presuming you already did that in the testing phase above, you've already completed this step. If you didn't make them executable earlier, do so now via instructions above under Testing Your Rules.

5. Setup the service. This will be OK because the scripts flush the caches.

```
systemctl start rules
systemctl enable rules
systemctl stop rules
```

You're done! Reboot and confirm your new rules and routes are functioning as expected. If not, review steps above and try again. If they still don't work, review Testing Your Rules.

References

Andreasson, Oskar. *Iptables Tutorial 1.2.1*. Chapter 6: Traversing of tables and chains. https://www.frozentux.net/iptables-tutorial/chunkyhtml/c962.html.

Andreasson, Oskar. 2006. *Iptables Tutorial 1.2.2.* http://homes.di.unimi.it/sisop/qemu/iptables-tutorial.pdf.

Anicas, Mitchell. 14 August 2015. *How To List and Delete Iptables Firewall Rules*. *Digital Ocean*, https://www.digitalocean.com/community/tutorials/how-to-list-and-delete-iptables-firewall-rules.

Ayuso, Pablo Neira. 2006. *Netfilter's Connection Tracking System*. ;LOGIN: The USENIX Magazine. Vol. 31, No. 3. *USENIX: The Advanced Computing Systems Association*, https://www.usenix.org/system/files/login/articles/892-neira.pdf.

best way to clear all iptables rules. 11 November 2010. Stack Exchange. *Server Fault*, https://serverfault.com/questions/200635/linux-iptables-best-way-to-clear-all-rules-leaving-anything-open.

Baturin, Daniil. (n.d.). *Overview [sic] ip2route cheatsheet*. http://baturin.org/docs.

Brown, Martin A. 26 April 2003. *Guide to IP Layer Network Administration with Linux*. Chapter 10: Advanced IP Routing. Version 0.4.4. *Linux-IP*, http://linux-ip.net.

Control Network Traffic with iptables. 28 February 2017. Linode. Weblog. *Linode*, https://linode.com/docs/security/firewalls/control-network-traffic-with-iptables.

Create table in iptables. 6 July 2013. Stack Exchange. *Ask Ubuntu*, https://askubuntu.com/questions/316990/create-table-in-iptables.

Differences Between IPTables and IPChains. 26 June 2007. *Red Hat Enterprise Linux Deployment Guide.* Red Hat, https://www.centos.org/docs/5/html/5.1/Deployment_Guide/s1-iptables-differences.html.

Edwards, N., & Engelhardt, J. (2013, January 15). *GMANE Mailing List Archive. GMANE*, http://comments.gmane.org/gmane.comp.security.firewalls.netfilter.general/45564.

Egan, David. 7 August 2016. *Persistent Iptables Rules in Ubuntu 16.04 Xenial Xerus*. Weblog. Dev Notes, http://dev-notes.eu/2016/08/persistent-iptables-rules-in-ubuntu-16-04-xenial-xerus.

Ellingwood, Justin. (20 August 2015). *A Deep Dive into Iptables and netfilter Architecture*. *Digital Ocean*, https://www.digitalocean.com/community/tutorials/a-deep-dive-into-iptables-and-netfilter-architecture.

Engelhardt, Jan. 28 February 2014. *Packet Flow in Netfilter and General Networking*. *Wikipedia*, https://upload.wikimedia.org/wikipedia/commons/3/37/Netfilter-packet-flow.svg.

Flickenger, Rob. (n.d.). *Hack #49. Using Custom Chains in iptable*s. Weblog. *O'Reilly*, https://www.safaribooksonline.com/library/view/linux-server-hacks/0596004613/ch04s06.html.

Force Torrent Traffic through VPN Split Tunnel Debian 8 + Ubuntu 16.04. (n.d.). *HTPC Guides*, https://www.htpcguides.com/force-torrent-traffic-vpn-split-tunnel-debian-8-ubuntu-16-04.

Gheorghe. Lucian. 15 December 2010. "Chapter 4: Packet Mangling with iptables." *Designing and Implementing Linux Firewalls and QoS using netfilter, iproute2, NAT and l7*-filter. pp. 113-117. *Cross Linux From Scratch*, http://clfs.org/~kb0iic/Linux%20Firewalls%20and%20QOS.pdf.

Hagen, Phil. 30 April 2019 (edited). *iptables Processing Flowchart*. Weblog. *Phil Hagen's Scratch Pad*, https://stuffphilwrites.com/2014/09/iptables-processing-flowchart.

Hambourg, Pascal. 2009. *NAT using iproute2*. Email. NARKIVE Newsgroup Archive. https://comp.os.linux.networking.narkive.com/mhXc9nQe/nat-using-iproute2#post6.

Hubert, Bert; Graf, Thomas; Larroy, Pedro Maxwell, Gregory; Schroeder, Paul B; Spaans, Jasper; van Mook, Remco; and van Oosterhout, Martijn. "Chapter 4: Rules - routing policy database." 22 July 2002. *Linux Advanced Routing & Traffic Control HOWTO*. Version 1.1. *The Linux Documentation Project*, http://lartc.org/howto/lartc.rpdb.html.

iproute2. (16 November 2017). *Wikipedia*, https://en.wikipedia.org/wiki/Iproute2.

IPROUTE2 Utility Suite Howto. (n.d.). *PolicyRouting.org*, http://www.policyrouting.org/iproute2.doc.html.

Iptables. 2008. *Chapter 6: Traversing of tables and chains*. http://www.iptables.info.

iptables - Target to route packet to specific interface? 29 December 2011. *Server Fault (Stack Exchange)*, http://serverfault.com/questions/345111/iptables-target-to-route-packet-to-specific-interface.

iptables vs route. 2 May 2012. Stack Exchange. *Superuser*, https://superuser.com/questions/419659/iptables-vs-route.

IptablesHowTo. 9 October 2017. Edited by Connor Imes. *Ubuntu Documentation*, https://help.ubuntu.com/community/IptablesHowTo.

Iptables MAC Address Filtering. 27 December 2005. Weblog. *NixCraft*, https://www.cyberciti.biz/tips/iptables-mac-address-filtering.html.

IPTables packet traverse map. 5 September 2011. *Admins ehow*, http://www.adminsehow.com/2011/09/iptables-packet-traverse-map.

iptables Release of the netfilter/iptables project. (27 May 2019). *NetFilter*, https://www.netfilter.org/projects/iptables/downloads.html

Iptables Tutorial 1.2.1. (n.d.). https://www.frozentux.net/iptables-tutorial/chunkyhtml/c3965.html.

Kernel Packet Traveling Diagram. (n.d.). <https://www.docum.org/docum.org/kptd/>.

Knome (editor). 13 December 2013. IptablesHowTo. <https://help.ubuntu.com/community/IptablesHowTo>.

Kuznetsov, Alexey. N. (14 April 1999). *IP Command Reference*. Moscow. *Institute for Nuclear Research*, https://cs.hofstra.edu/~cscccl/csc175/ip-cref.pdf.

Linux 2.6 - man page for iptables (linux section 8). (n.d.). *Unix & Linux Forums*, https://www.unix.com/man-page/linux/8/iptables.

Load balancing using iptables with CONNMARK. (n.d.). SystemRescueCd. http://www.system-rescue-cd.org/networking/Load-balancing-using-iptables-with-connmark.

Marsh, Matthew G. *Policy Routing Using Linux*. Sams, 2001. *PolicyRouting.org*, http://www.policyrouting.org/PolicyRoutingBook.

Mitasch, Christoph. 10 June 2015. *Saving Iptables Firewall Rules Permanently*. Weblog. https://www.thomas-krenn.com/en/wiki/Saving_Iptables_Firewall_Rules_Permanently.

Morris, James. security: add iptables "security" table for MAC rules. 29 January 2008. E-mail. *LWN.net*, https://lwn.net/Articles/267140.

Mortimer, Ian. 21 September 2000. *rp_filter not working ?? NetFilter Mailing Lists*, http://lists.netfilter.org/pipermail/netfilter/2000-September/005400.html.

netfilter. (20 December 2017). *Wikipedia*, https://en.wikipedia.org/wiki/netfilter.

nftables. (10 December 2017). *Wikipedia*, https://en.wikipedia.org/wiki/Nftables.

Only allow certain outbound traffic on certain interfaces. 26 April 2011. Stack Exchange. *Unix & Linux*, http://unix.stackexchange.com/questions/12085/only-allow-certain-outbound-traffic-on-certain-interfaces.

Packet flow in netfilter and General Networking. Image. *Wikipedia*, https://upload.wikimedia.org/wikipedia/commons/3/37/netfilter-packet-flow.svg.

Redirect Routing on Non-Default Interface. 12 December 2008. *Ubuntu forums*, http://ubuntuforums.org/showthread.php?t=1008620.

Russell, Rusty. 29 July 2001. *Linux Networking-concepts HOWTO*. *NetFilter Mailing Lists*, http://www.netfilter.org/documentation/HOWTO//networking-concepts-HOWTO.html.

Russell, Rusty; Welte, Harold. 7 February 2002. Linux netfilter Hacking HOWTO. Weblog. https://www.netfilter.org/documentation/HOWTO/netfilter-hacking-HOWTO.html.

*Setup routing and iptables for new VPN connection to redirect **only** ports 80 and 443*. 7 November 2011. *Superuser*, http://superuser.com/questions/354855/setup-routing-and-iptables-for-new-vpn-connection-to-redirect-only-ports-80.

Simon. 28 August 2011. *MAKING ALL NETWORK TRAFFIC FOR A LINUX USER USE A SPECIFIC NETWORK INTERFACE*. *Niftiest Software*, http://www.niftiestsoftware.com/2011/08/28/making-all-network-traffic-for-a-linux-user-use-a-specific-network-interface.

Split tunnel routing a specific port over OpenVPN on Ubuntu Server 12.04. 11 June 2013. *Superuser*, http://superuser.com/questions/606159/split-tunnel-routing-a-specific-port-over-openvpn-on-ubuntu-server-12-04.

Suehring, Steve; Ziegler, Robert. 14 September 2005. *Linux Firewalls*. 3rd edition. Novell Press.

Summary of changes from v2.6.8 to v2.6.9. 19 October 2004. Changelog. https://mirrors.edge.kernel.org/pub/linux/kernel/v2.6/ChangeLog-2.6.9.

The netfilter.org "nftables" project. (n.d.). *Netfilter*, https://netfilter.org/projects/nftables.

UFW. 31 March 2017. *Ubuntu Documentation*, https://help.ubuntu.com/community/UFW.

Gite, Vivek. *Understanding Routing Table.* 13 March 2008. *Nixcraft*, https://www.cyberciti.biz/faq/what-is-a-routing-table.

Wagner, Tyler J. 13 July 2009. *Disabling Reverse-Path Filtering in Complex Networks*. http://www.tolaris.com/2009/07/13/disabling-reverse-path-filtering-in-complex-networks.

Westphal, Florian. (2017). *improvements to conntrack table overflow handling*. Montreal, Canada. *Netdev 2.1, The Technical Conference on Linux Networking,* https://www.netdevconf.org/2.1/papers/conntrack.pdf.

What is a Subnet Mask? 13 November 2018. iplocation.net. https://www.iplocation.net/subnet-mask.

What is the difference between -m conntrack --ctstate and -m state --state. 7 January 2014. Stack Exchange. *Unix & Linux*, https://unix.stackexchange.com/questions/108169/what-is-the-difference-between-m-conntrack-ctstate-and-m-state-state.

xenial (8) iptables.8.gz. (n.d.). *Ubuntu manuals*, http://manpages.ubuntu.com/manpages/xenial/man8/iptables.8.html.

xenial (8) iptables-extensions.8.gz. (n.d.). *Ubuntu manuals*, http://manpages.ubuntu.com/manpages/xenial/man8/iptables-extensions.8.html

zesty(8) nft.8.gz. 10 October 2016. *Ubuntu Man Pages*, http://manpages.ubuntu.com/manpages/xenial/en/man8/nft.8.html.

Endnotes

[1] ip6tables is the IPv6 equivalent of iptables. This document is limited to discussing *iptables*, which only covers IPv4.

[2] The Address Resolution Protocol (ARP) packet filter rules in the Linux kernel firewall modules. Not discussed in this document.

[3] *ebtables* is a filtering tool that enables transparent filtering of network traffic passing through a Linux bridge. It is beyond the scope of this document.

[4] Source: *The Swift Programming Language (Swift 4.1)*. © Apple, Inc. Retrieved March 24, 2018 from https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/AdvancedOperators.html.

[5] *Quality of Service* (QoS) refers to the proactive control and management of network resources by setting priorities for specific types of data on a network. It is a general term, and not a protocol or IP characteristic. It is essentially a methodology.

[6] *Type of Service* (ToS) is a byte in the IP header of a packet that classifies the *precedence* and *type* of traffic of the packet. It carries very specific meanings and will cause a router to prioritize the packet ahead of other, non-declared or lower priority declaration packets. ToS bytes are an advanced method of forcing data packet prioritization.

[7] James Morris announced the Security table in *iptables* via a Request For Comment email in January 2008 where he posited moving some marking features of the Mangle table to this new Security table. Thankfully, that never happened as most Linux distributions do not use SELinux, which is required for the Security table to function (Morris, 2008).

[8] Only relevant when using iptables command line (i.e. not in a script). Applies to specified table (default is Filter table).

[9] User-created chains are temporary and will disappear on reboot. Think of them as extensions of a table (that is where they reside). User-defined chains do not behave the same as static (default) chains. They are simply collections of rules.

[10] As of *iptables* version 1.6.0, only one argument is possible per -p parameter. Branching on more than one protocol requires an additional line of code for each additional protocol.

[11] *Port Forwarding* (also known as p*ort mapping*) is an act of Network Address Translation (NAT) that redirects in incoming network communication to particular internal IP address and port number on the associated destination device. It allows granular routing of network traffic based on either a specific incoming port number, or a combination of destination IP address and port number.