LINUX FIREWALLS

ATTACK DETECTION AND RESPONSE WITH IPTABLES, PSAD, AND FWSNORT

Linux Firewalls *is a great book.* ***—From the foreword by Richard Bejtlich of TaoSecurity.com***

PRAISE FOR *LINUX FIREWALLS*

"Right from the start, the book presented valuable information and pulled me in. Each of the central topics were thoroughly explained in an informative, yet engaging manner. Essentially, I did not want to stop reading." –SLASHDOT

"What really makes this book different from the others I've seen over the years is that the author approaches the subject in a layered method while exposing potential vulnerabilities at each step. So for those that are new to the security game, the book also takes a stab at teaching the basics of network security while teaching you the tools

to build a modern firewall." –INFOWORLD

"This admirable, eminently usable text goes much further than advertised." –LINUX USER AND DEVELOPER

"This well-researched book heightens an average system administrator's awareness to the vulnerabilities in his or her infrastructure, and the potential to find hardening solutions." –FREE SOFTWARE MAGAZINE

"If you or anyone you know is responsible for keeping a secure network, *Linux Firewalls* is an invaluable resource to have by your side." –LINUXSECURITY.COM

"If you're building a Linux firewall and want to know what all the bells and whistles are, when you might want to set them off, and how to hook them together, here you go." –;LOGIN

"If you run one or more Linux based firewalls, this book will not only help you to configure them securely, it will help you understand how they can be monitored to discover evidence of probes, abuse and denial of service attacks." –RON GULA, CTO & CO-FOUNDER OF TENABLE NETWORK SECURITY

LINUX FIREWALLS

Attack Detection and Response with iptables, psad, and fwsnort

by Michael Rash

®

San Francisco

fire_TITLE_COPY.fm Page iv Monday, April 14, 2008 10:48 AM

Printed on recycled paper in the United States of America

11 10 09 08 2 3 4 5 6 7 8 9

ISBN-10: 1-59327-141-7
ISBN-13:
978-1-59327-141-1

Publisher: William Pollock Production Editor: Christina
Samuell Cover and Interior Design: Octopod Studios
Developmental Editor: William Pollock Technical
Reviewer: Pablo Neira Ayuso Copyeditors: Megan
Dunchak and Bonnie Granat Compositors: Christina
Samuell and Riley Hoffman Proofreaders: Karol Jurado
and Riley Hoffman Indexer: Nancy Guenther

For information on book distributors or translations, please contact No Starch Press, Inc. directly:

No Starch Press, Inc. 555 De Haro Street, Suite 250, San Francisco, CA 94107 phone:

415.863.9900; fax: 415.863.9950; info@nostarch.com; www.nostarch.com

Library of Congress Cataloging-in-Publication Data

Rash,
Michael.
Linux firewalls : attack detection and response with iptables, psad, and fwsnort / Michael Rash.
p. cm. Includes index. ISBN-13: 978-1-59327-141-1 ISBN-10: 1-59327-141-7 1. Computers--Access control. 2. Firewalls (Computer security) 3. Linux. I. Title. QA76.9.A25R36 2007 005.8--dc22
20060266
79

To Katie and little Bella

BRIEF CONTENTS

Acknowledgments xv

Foreword by Richard Bejtlich xvii

Introduction 1

Chapter 1: Care and Feeding of iptables 9

Chapter 2: Network Layer Attacks and Defense 35

Chapter 3: Transport Layer Attacks and Defense 49

Chapter 4: Application Layer Attacks and Defense.....69

Chapter 5: Introducing psad: The Port Scan Attack Detector 81

Chapter 6: psad Operations: Detecting Suspicious Traffic 99

Chapter 7: Advanced psad Topics: From Signature Matching to OS Fingerprinting.....113

Chapter 8: Active Response with psad.....131

Chapter 9: Translating Snort Rules into iptables Rules 149

Chapter 10: Deploying fwsnort.....173

Chapter 11: Combining psad and fwsnort 193

Chapter 12: Port Knocking vs. Single Packet Authorization 213

Chapter 13: Introducing fwknop 231

Chapter 14: Visualizing iptables Logs.....257

Appendix A: Attack Spoofing 279

Appendix B: A Complete fwsnort Script 285

Index 291

CONTENTS IN DETAIL

ACKNOWLEDGMENTS xv

FOREWORD by Richard Bejtlich xvii

INTRODUCTION 1

Why Detect Attacks with iptables? ..... 2 What About Dedicated Network Intrusion Detection Systems? ..... 3 Defense in Depth ..... 4 Prerequisites

..... 4 Technical
References ..... 5 About the
Website ..... 5 Chapter
Summaries 6

1CARE AND FEEDING OF IPTABLES 9

iptables ..... 9 Packet
Filtering with iptables ..... 10 Tables
..... 11 Chains
..... 11 Matches
..... 12 Targets
..... 12 Installing iptables
..... 12 Kernel Configuration
..... 14 Essential Netfilter
Compilation Options ..... 15 Finishing the Kernel
Configuration ..... 16 Loadable Kernel Modules vs.
Built-in Compilation and Security ..... 16 Security and Minimal Compilation
..... 17 Kernel Compilation and Installation
..... 18 Installing the iptables Userland Binaries
..... 19 Default iptables Policy
..... 20 Policy Requirements
..... 20 iptables.sh Script Preamble
..... 22 The INPUT Chain
..... 22 The OUTPUT Chain
..... 24 The FORWARD Chain
..... 25 Network Address Translation
..... 26 Activating the Policy
..... 27 iptables-save and iptables-restore
..... 27 Testing the Policy: TCP
..... 29 Testing the Policy: UDP
..... 31 Testing the Policy: ICMP
..... 32 Concluding Thoughts
..... 33

2NETWORK LAYER ATTACKS AND DEFENSE 35

Logging Network Layer Headers with iptables 35
Logging the IP Header ..... 36 Network Layer
Attack Definitions ..... 38 Abusing the Network
Layer ..... 39 Nmap ICMP Ping
..... 39 IP Spoofing
..... 40 IP Fragmentation
..... 41 Low TTL Values
..... 42 The Smurf Attack
..... 43 DDoS Attacks
..... 44 Linux Kernel IGMP Attack
..... 44 Network Layer Responses
..... 45 Network Layer Filtering
Response ..... 45 Network Layer Thresholding

Response ..... 45 Combining Responses Across Layers
..... 46

3 TRANSPORT LAYER ATTACKS AND DEFENSE 49

Logging Transport Layer Headers with iptables 50
Logging the TCP Header ..... 50 Logging the
UDP Header ..... 52 Transport Layer Attack
Definitions ..... 52 Abusing the Transport
Layer ..... 53 Port Scans
..... 53 Port Sweeps
..... 61 TCP Sequence Prediction
Attacks ..... 61 SYN Floods
..... 62 Transport Layer Responses
..... 62 TCP Responses
..... 62 UDP Responses
..... 66 Firewall Rules and Router
ACLs 67

4 APPLICATION LAYER ATTACKS AND DEFENSE 69

Application Layer String Matching with iptables 70
Observing the String Match Extension in Action ..... 70 Matching
Non-Printable Application Layer Data ..... 71 Application Layer
Attack Definitions ..... 72 Abusing the
Application Layer ..... 73 Snort Signatures
..... 74 Buffer Overflow Exploits
..... 74 SQL Injection Attacks
..... 76 Gray Matter Hacking
..... 77 Encryption and Application
Encodings ..... 79 Application Layer Responses
..... 80

5 INTRODUCING PSAD: THE PORT SCAN ATTACK DETECTOR 81

History ..... 81 Why
Analyze Firewall Logs? ..... 82 psad
Features ..... 83 psad
Installation ..... 83 psad
Administration ..... 85 Starting and
Stopping psad ..... 85 Daemon Process
Uniqueness ..... 86 iptables Policy Configuration
..... 86 syslog Configuration
..... 88 whois Client
..... 89 psad Configuration

..... 90 /etc/psad/psad.conf
..... 90 /etc/psad/auto_dl
..... 96 /etc/psad/signatures
..... 96 /etc/psad/snort_rule_dl
..... 97 /etc/psad/ip_options
..... 97 /etc/psad/pf.os
..... 97 Concluding Thoughts
..... 98

6 PSAD OPERATIONS: DETECTING SUSPICIOUS TRAFFIC 99

Port Scan Detection with psad ..... 100 TCP connect() Scan ..... 101 TCP SYN or Half-Open Scan ..... 103 TCP FIN, XMAS, and NULL Scans ..... 105 UDP Scan ..... 106 Alerts and Reporting with psad ..... 108 psad Email Alerts ..... 108 psad syslog Reporting ..... 110 Concluding Thoughts 112

7 ADVANCED PSAD TOPICS: FROM SIGNATURE MATCHING TO OS FINGERPRINTING 113

Attack Detection with Snort Rules ..... 113 Detecting the ipEye Port Scanner ..... 115 Detecting the LAND Attack ..... 116 Detecting TCP Port 0 Traffic ..... 116 Detecting Zero TTL Traffic ..... 117 Detecting the Naptha Denial of Service Attack ..... 117 Detecting Source Routing Attempts ..... 118 Detecting Windows Messenger Pop-up Spam ..... 118 psad Signature Updates ..... 119 OS Fingerprinting ..... 120 Active OS Fingerprinting with Nmap ..... 120 Passive OS Fingerprinting with p0f 121

DShield Reporting ..... 123 DShield Reporting Format ..... 124 Sample DShield Report ..... 124 Viewing psad Status Output ..... 124 Forensics Mode ..... 128 Verbose/Debug Mode ..... 128 Concluding Thoughts 130

8 ACTIVE RESPONSE WITH PSAD 131

Intrusion Prevention vs. Active Response 131
Active Response Trade-offs 133
Classes of Attacks ..... 133 False Positives
..... 134 Responding to Attacks with
psad ..... 134 Features
..... 135 Configuration Variables
..... 135 Active Response Examples
..... 137 Active Response Configuration
Settings ..... 138 SYN Scan Response
..... 139 UDP Scan Response
..... 140 Nmap Version Scan
..... 141 FIN Scan Response
..... 141 Maliciously Spoofing a Scan
..... 142 Integrating psad Active Response with
Third-Party Tools ..... 143 Command-Line Interface
..... 143 Integrating with Swatch
..... 145 Integrating with Custom Scripts
..... 146 Concluding Thoughts
..... 147

9 TRANSLATING SNORT RULES INTO IPTABLES RULES 149

Why Run fwsnort? 150
Defense in Depth ..... 151 Target-Based
Intrusion Detection and Network Layer Defragmentation ..... 151 Lightweight Footprint
..... 152 Inline Responses
..... 152 Signature Translation Examples
..... 153 Nmap command attempt Signature
..... 153 Bleeding Snort “Bancos Trojan” Signature
..... 154 PGPNet connection attempt Signature
..... 154 The fwsnort Interpretation of Snort Rules
..... 155 Translating the Snort Rule Header
..... 155 Translating Snort Rule Options: iptables Packet
Logging ..... 157 Snort Options and iptables Packet Filtering
..... 160 Unsupported Snort Rule Options
..... 171 Concluding Thoughts
..... 172

10 DEPLOYING FWSNORT 173

Installing fwsnort 173

Running fwsnort 175
Configuration File for fwsnort ..... 177 Structure of
fwsnort.sh ..... 179 Command-Line Options
for fwsnort ..... 182 Observing fwsnort in Action
..... 184 Detecting the Trin00 DDoS Tool
..... 184 Detecting Linux Shellcode Traffic
..... 185 Detecting and Reacting to the Dumador Trojan
..... 186 Detecting and Reacting to a DNS Cache-Poisoning Attack
..... 188 Setting Up Whitelists and Blacklists
..... 191 Concluding Thoughts
..... 192

11 COMBINING PSAD AND FWSNORT 193

Tying fwsnort Detection to psad Operations 194
WEB-PHP Setup.php access Attack ..... 194 Revisiting
Active Response ..... 198 psad vs.
fwsnort ..... 198 Restricting psad
Responses to Attacks Detected by fwsnort ..... 199 Combining fwsnort and
psad Responses ..... 199 DROP vs. REJECT Targets
..... 201 Thwarting Metasploit Updates
..... 204 Metasploit Update Feature
..... 204 Signature Development
..... 206 Busting Metasploit Updates with
fwsnort and psad ..... 208 Concluding Thoughts
..... 212

12 PORT KNOCKING VS. SINGLE PACKET AUTHORIZATION 213

Reducing the Attack Surface ..... 213 The
Zero-Day Attack Problem ..... 214 Zero-Day
Attack Discovery ..... 215 Implications for
Signature-Based Intrusion Detection ..... 215 Defense in Depth
..... 216 Port Knocking
..... 217 Thwarting Nmap and
the Target Identification Phase ..... 218 Shared Port-Knocking
Sequences ..... 218 Encrypted Port-Knocking
Sequences ..... 221 Architectural Limitations of Port
Knocking ..... 223 Single Packet Authorization
..... 226 Addressing Limitations of Port
Knocking ..... 227 Architectural Limitations of SPA
..... 228 Security Through Obscurity?
..... 229 Concluding Thoughts
..... 230

13 INTRODUCING FWKNOP 231

fwknop Installation ..... 232 fwknop Configuration ..... 234 /etc/fwknop/fwknop.conf ..... 234 /etc/fwknop/access.conf ..... 237 Example /etc/fwknop/access.conf File ..... 240 fwknop SPA Packet Format ..... 241 Deploying fwknop ..... 243 SPA via Symmetric Encryption ..... 244 SPA via Asymmetric Encryption ..... 246 Detecting and Stopping a Replay Attack ..... 249 Spoofing the SPA Packet Source Address ..... 251 fwknop OpenSSH Integration Patch ..... 252 SPA over Tor ..... 254 Concluding Thoughts 255

14 VISUALIZING IPTABLES LOGS 257

Seeing the Unusual ..... 258 Gnuplot ..... 260 Gnuplot Graphing Directives ..... 260 Combining psad and Gnuplot ..... 261 AfterGlow ..... 262 iptables Attack Visualizations ..... 263 Port Scans ..... 264 Port Sweeps ..... 267 Slammer Worm ..... 270 Nachi Worm ..... 272 Outbound Connections from Compromised Systems ..... 273 Concluding Thoughts 277

A ATTACK SPOOFING 279

Connection Tracking ..... 280 Spoofing exploit.rules Traffic ..... 282 Spoofed UDP Attacks 283

B A COMPLETE FWSNORT SCRIPT 285

INDEX 291

ACKNOWLEDGMENTS

Linux Firewalls was made possible with the help of a host of folks at every step along the way. I'd particularly like to thank the people at No Starch Press for the efforts they put forth. William Pollock, Bonnie Granat, Megan Dunchak, and Christina Samuell all contributed many hours of expert editing, and the book is higher quality as a result. To Pablo Neira Ayuso, thanks for helping to make Netfilter and iptables what they are today, and for handling the technical edit of the material in this book. Ron Gula, CTO of Tenable Network Security, and Raffael Marty, chief security strategist of Splunk, both contributed constructive criticism, and they were kind enough to endorse the book before it was published. I also wish to thank Richard Bejtlich, founder of TaoSecurity, for writing an excellent foreword. Richard, your books are an inspiration. My parents, James and Billie Mae, and my brother, Brian, all deserve a special thank you for their constant encouragement. Finally, many thanks go to my wife, Katie. This book would not have been possible without you.

FOREWORD

When hearing the term *firewall*, most people think of a product that inspects network traffic at the network and transport layers of the OSI Reference Model and makes pass or filter decisions. In terms of products, dozens of firewall types exist. They are differentiated by the data source they inspect (e.g., network traffic, host processes, or system calls) and the depth to which they inspect those sources. Almost any device that inspects communi- cation and decides whether to pass or filter it could be considered a firewall product.

Marcus Ranum, inventor of the proxy firewall and the implementer of the first commercial firewall product, offered a definition of the term *firewall* in the mid-1990s when he said, "A firewall is the implementation of your Internet security policy." [1] This is an excellent definition because it is product- neutral, timeless, and realistic. It applies equally well to the original firewall book, *Firewalls and Internet Security* by William R. Cheswick and Steven M. Bellovin (Addison-Wesley Professional, 1994), as it does to the book you're reading now.

[1] *Computer Security Journal*, Vol. XI, No. 1, Spring 1995 (http://www.spirit.com/CSI/Papers/hownot.htm)

In the spirit of Ranum's definition, a firewall could also be considered a policy enforcement system.

Devices that inspect and then pass or filter network traffic could be called *network policy enforcement systems*. Devices that inspect and then pass or filter host-centric activities could be called *host policy enforce- ment systems*. In either case, emphasis on policy enforcement focuses attention on the proper role of the firewall as a device that implements policy instead of one that just "stops bad stuff."

With respect to "bad stuff," it's reasonable to ask if firewalls even matter in today's enterprise. Properly configured traditional network firewall pro- ducts basically deny all but allowed Internet protocols, IP addresses, TCP/UDP ports, and ICMP types and codes. In the modern attack environment, this sort of defense is entirely insufficient. Restricting those exploitation channels is necessary to restrict the ingress and egress paths to a target, but network and transport layer filtering has been a completely inadequate counter- measure for at least a decade.

In 2007, the most effective way to compromise a client is to entice the user to activate a malicious executable, send the user a link that hosts malicious content, or attack another client-side component of the user's computing experience. In many cases, exploitation doesn't rely on a vulnerability that could be patched or a configuration that could be tightened. Rather, attackers exploit weaknesses in rich-media platforms like JavaScript and Flash, which are increasingly required for browsing the Web today.

In 2007, the most effective way to compromise a server is to avoid the operating system and exploit the application. Web applications dominate the server landscape, and they are more likely to suffer from architectural and design flaws than from vulnerabilities that can be patched. In the late 1990s, it was fashionable to change the prices for the items in one's shopping cart to demonstrate insecure web applications. Thanks to Ajax, almost a decade later the shopping cart is running on the client and users are again changing prices—and worse.

All of this makes the picture seem fairly bleak for firewall products. Many have adapted by incorporating deep packet inspection or operating at or beyond the application layer of the OSI Reference Model. Others operate as *intrusion prevention systems*, using a clever marketing term to differentiate themselves in a seemingly commoditized market. Is there a role for firewalls, especially open source products, in the age of client-side attacks and web application exploitation?

The answer is yes—and you are reading one approach right now. Michael Rash is a pioneer in the creative use of network technologies for defensive purposes. The security research and development world tends to be dominated by offensive tools and techniques, as a quick glance at the speakers list for a certain Las Vegas hacker convention will demonstrate. Bucking this trend, Michael continues to invent and improve upon methods for protecting assets from attack. After getting a look at the dark side at an offensive conference, almost all of us return to the seemingly mundane job of protecting our enterprises. Thanks to this book, we have an additional suite of programs and methods to make our jobs easier.

While reading a draft of this book, I identified a few themes. First, host-centric defense is increasingly important as devices become self-reliant and are exposed to the Internet. An extreme example of this evolution is the introduction of IPv6, which when

deployed as intended by its progenitors restores the "end-to-end" nature of the original Internet. Of course, *end-to-end* can be translated into *attacker-to-victim*, so additional ways for hosts to protect themselves are appreciated. *Linux Firewalls* will teach you how hosts can pro- tect themselves using host-based firewalls and tools.

Second, despite the fact that hosts must increasingly defend themselves, host-centric measures alone are inadequate. Once a host has been compro- mised, it can no longer be responsible for its own defenses. Upon breaching a system, intruders routinely disable host firewalls, antivirus software, and other protective agents. Therefore, network-centric filtering devices are still required wherever possible. An endpoint controlled by a victim can only use the communication channels allowed by the network firewall, at least limiting the freedom to maneuver enjoyed by the intruder. *Linux Firewalls* will also teach you how network devices can protect hosts.

Third, we must look at creative ways to defend our assets and understand the attack landscape. Single Packet Authorization is a giant step beyond port knocking if one wants to limit access to sensitive services. Visualization helps render logs and traffic in a way that enables analysts to detect subtle events of interest. After reading this book, you may find additional ways to leverage your defensive infrastructure not anticipated by others, including the author.

I'd like to conclude these thoughts by speaking as a book reviewer and author. Between 2000 and mid-2007, I've read and reviewed nearly 250 tech- nical books. I've also written several books, so I believe I can recognize a great book when I see it. *Linux Firewalls* is a great book. I'm a FreeBSD user, but *Linux Firewalls* is good enough to make me consider using Linux in certain circumstances! Mike's book is exceptionally clear, organized, concise, and actionable. You should be able to read it and implement everything you find by following his examples. You will not only familiarize yourself with tools and learn to use techniques, but you will be able to appreciate the author's keen defensive insights.

The majority of the world's digital security professionals focus on defense, leaving offense to the bad guys, police, and military. I welcome books like *Linux Firewalls* that bring real defensive tools and techniques to the masses in a form that can be digested and deployed for minimum cost and effort. Good luck—we all need it.

Richard Bejtlich Director of Incident Response,
General Electric Manassas Park, VA

INTRODUCTION

The offense seems to be getting the upper hand. Rarely a day goes by without news of a new exploit for a software vulnerability, a more effective method of distributing spam (my inbox can attest to this), or a high-profile theft of sensitive personal data from a corporation or government agency. Achieving secure computing is a perpetual challenge. There is no shortage of technologies designed to foil crafty black hats, and yet they continue to successfully compromise systems and networks.

For every class of security problem, there is almost certainly either an open source or proprietary solution designed to combat it. This is parti- cularly true in the areas of network intrusion detection systems and network access control devices—firewalls, filtering routers, and the like. A trend in firewall technology is to combine application layer inspection techniques from the intrusion detection world with the ability to filter network traffic, something firewalls have been doing for a long time. It is the goal of this book to show that the iptables firewall on Linux systems is well positioned to take advantage of this trend, especially when it is combined with some additional software designed to leverage iptables from an intrusion detection standpoint.

It is my hope that this book is unique in the existing landscape of pub- lished works. There are several

excellent books out there that discuss various aspects of Linux firewalls, but none to my knowledge that concentrate specifically on attacks that can be detected (and in some cases thwarted) by iptables and the data it provides. There are also many books on the topic of intrusion detection, but none focuses on using firewalling technology to truly supplement the intrusion detection process. This book is about the convergence of these two technologies.

I will devote significant coverage to three open source software projects that are designed to maximize the effectiveness of iptables for attack detection and prevention. These are the projects:

psad An iptables log analyzer and active response tool **fwsnort** A script that translates Snort rules into equivalent iptables rules **fwknop** An implementation of Single Packet Authorization (SPA) for iptables

All of these projects are released as open source software under the GNU Public License (GPL) and can be downloaded from http:// www.cipherdyne.org.

Why Detect Attacks with iptables?

> ROSENCRANTZ: I mean, what exactly do you *do*?
>
> PLAYER: We keep to our usual stuff, more or less, only inside out. We do on stage the things that are supposed to happen off. Which is a kind of integrity, if you look on every exit being an entrance somewhere else.
>
> —Tom Stoppard, *Rosencrantz & Guildenstern Are Dead*

If you run the Linux operating system, you have likely encountered the iptables firewall. This is for good reason, as iptables provides an effective means to control who talks to your Linux system over a network connection and how they do it. In the vast uncontrolled network that is the Internet, attacks can herald from just about any corner of the globe—even though the perpetrator might physically be located in the next state (or the next room). If you run a networked Linux machine, your system is at risk of being attacked and potentially compromised every second of every day.

Deploying a strict iptables filtering policy is a good first step toward main- taining a strong security stance. Even if your Linux system is connected to a network that is protected upstream by another firewall or other filtering device, there is always a chance that this upstream device may be unable to provide adequate protection. Such a device might be configured improperly, it might suffer from a bug or other failure, or it might not possess the ability to protect your Linux system from certain classes of attack. It is important to achieve a decent level of redundancy wherever possible, and the security benefits of run- ning iptables on every Linux system (both servers and desktops) can outweigh

the additional management overhead. Put another way, the risks of a compro- mise

and the value of the data that could be lost will likely outweigh the cost of deploying and maintaining iptables throughout your Linux infrastructure.

The primary goal of this book is to show you how to maximize iptables from the standpoints of detecting and responding to network attacks. A restrictive iptables policy that limits who can talk to which services on a Linux system is a good first step, but you will soon see that you can take things much further.

What About Dedicated Network Intrusion Detection Systems?

The job of detecting intrusions is usually left to special systems that are designed for this purpose and that have a broad view of the local network. This book does not advocate changing this strategy. There is no substitute for having a dedicated network intrusion detection system (IDS) as a part of the security infrastructure charged with protecting a network. In addition, the raw packet data that an IDS can collect is an invaluable source of data. When- ever a security analyst is tasked with figuring out what happened during an attack or a system compromise, having the raw packet data is absolutely critical to piecing things together, and an event from an IDS can point the way. Without an IDS to call attention to suspicious activity, an analyst might never even suspect that a system is under attack.

What this book *does* advocate is using iptables to supplement existing intrusion detection infrastructures. The main focus of iptables is applying policy restrictions to network traffic, not detecting network attacks. However, iptables offers powerful features that allow it to emulate a significant portion of the capabilities that traditionally lie within the purview of intrusion detec- tion systems. For example, the iptables logging format provides detailed data on nearly every field of the network and transport layer headers (including IP and TCP options), and the iptables string matching capability can perform byte sequence matches against application layer data. Such abilities are critical for providing the ability to detect attempted intrusions.

Intrusion detection systems are usually passive devices that are not configured to automatically take any punitive action against network traffic that appears to be malicious. In general, this is for good reason because of the risk of misidentifying benign traffic as something more sinister (known as a *false positive*). However, some IDSes can be deployed inline to network traffic, and when deployed in this manner such a system is typically referred to as a network *intrusion prevention system (IPS)*.[1] Because iptables is a firewall, it is *always* inline to network traffic, which allows many attacks to be filtered out before they cause significant damage. Many organizations have been hesitant to deploy an inline IPS in their network infrastructure because of basic connectivity and performance concerns. However, in some circumstances having the ability to filter traffic based on application layer inspection criteria is quite useful, and on Linux systems, iptables can provide basic IPS function- ality by recasting IDS signatures into iptables policies to thwart network attacks.

[1] Despite the lofty-sounding name and the endless vendor marketing hype, a network intrusion prevention system would be nothing without a way to *detect* attacks—and the detection mechanisms

come from the IDS world. A network IPS usually just has some extra machinery to handle inline traffic and respond to attacks in this context.

Defense in

Depth

Defense in depth is a principle that is borrowed from military circles and is commonly applied to the field of computer security. It stipulates that attacks must be *expected* at various levels within an arbitrary system, be it anything from a computer network to a physical military installation. Nothing can ever ensure that attacks will never take place. Furthermore, some attacks may be successful and compromise or destroy certain components of a system. There- fore, it is important to employ multiple levels of defensive mechanisms at various levels within a system; where an attack compromises one security device, another device may succeed in limiting additional damage.

In the network security space, Snort is the champion of the open source intrusion detection world, and many commercial vendors have produced excellent firewalls and other filtering devices. However, if you are running Linux within your infrastructure, the real question is whether it is prudent to rely *solely* on these security mechanisms to protect your critical assets. The defense-in-depth principle indicates that iptables can serve as an important supplement to existing security infrastructures.

Prerequisites

This book assumes some familiarity with TCP/IP networking concepts and Linux system administration. Knowledge of the Open System Interconnec- tion (OSI) Reference Model and the main network and transport layer protocols (IPv4, ICMP, TCP, and UDP), as well as some knowledge of the DNS and HTTP application protocols would be most helpful. Although frequent references are made to the various layers of the OSI Reference Model, the network, transport, and application layers (3, 4, and 7, respectively) receive the vast majority of the discussion. The session and presentation layers are not covered, and the physical and data link layers are only briefly touched upon (comprehensive information on layer 2 filtering can be found at http://ebtables.sourceforge.net). The coverage of the network, transport, and application layers emphasizes attacks that are possible at each of these layers—knowledge of the structure and functionality at each of these layers is largely assumed. Even though wireless protocols and IPv6 are not specifically discussed, many of the examples in the book apply to these protocols as well.

A working knowledge of basic programming concepts (especially within the Perl and C programming languages) would also be useful, but code examples are generally broken down and explained. A few places in the book show raw packet data displayed via the tcpdump Ethernet sniffer, so some experience with an Ethernet sniffer such as tcpdump or Wireshark would be helpful. With the exception of the material described above, no prior knowl- edge of computer security, network intrusion detection, or firewall concepts is assumed.

Finally, this book concentrates on network attacks—detecting them and responding to them. As such, this book generally does not discuss host-level security issues such as the need to harden the system running iptables by removing compilers, severely curtailing user accounts, applying

the latest security patches, and so on. The Bastille Linux project (see http://www.bastille-linux.org) provides excellent information on host security issues, however. For the truly hard-core, the NSA SELinux distribution (see http://www.nsa.gov/selinux) is a stunning effort to increase system security starting with the component that counts the most—the kernel itself.

Technical References

The following titles are some excellent supporting references for the more technical aspects of this book:

Building Internet Firewalls, 2nd Edition; Elizabeth D. Zwicky, Simon Cooper, and D. Brent Chapman (O'Reilly, 2000) *Computer Networks, 4th Edition*; Andrew S. Tannenbaum (Prentice Hall PTR, 2002) *Firewalls and Internet Security: Repelling the Wily Hacker, 2nd Edition*; William R. Cheswick, Steven M. Bellovin, and Aviel D. Rubin (Addison-Wesley Professional, 2003) *Linux System Security, 2nd Edition*; Scott Mann and Ellen L. Mitchell (Pear- son Education, 2002) *Programming Perl, 3rd Edition*; Larry Wall, Tom Christiansen, and Jon Orwant (O'Reilly, 2000) *The Tao of Network Security Monitoring: Beyond Intrusion Detection*; Richard Bejtlich (Addison-Wesley Professional, 2004) *The TCP/IP Guide*; Charles M. Kozierok (No Starch Press, 2005) *TCP/IP Illustrated, Volume 1: The Protocols*; W. Richard Stevens (Addison- Wesley, 1994)

About the Website

Contained within this book are several example scripts, iptables policies and commands, and instances of network attacks and associated packet captures. All of these materials can also be downloaded from the book's companion website, which is available at http://www.cipherdyne.org/LinuxFirewalls. Having an electronic copy is the best way to tinker and experiment with the concepts and code yourself. Also available on the website are examples of the psad, fwsnort, and fwknop projects in action, along with documentation and the Trac interface (http://trac.edgewall.com), which enables you to view the source code for each project. The source code for each project is carefully archived within a Subversion repository (http://subversion.tigris.org) so that it is easy to visualize how the code changes from one version to the next. Finally, some interesting graphical representations of iptables log data can also be found on the website.

If you have questions while going through this book, you may also find answers on the book's website. Please don't hesitate to ask me any questions you may have regarding any of the material covered. You can reach me via email at mbr@cipherdyne.org.

Chapter Summaries

As you make your way through *Linux Firewalls*, you'll cover a lot of ground. This section gives you a brief overview of each chapter so you'll know what to expect.

Chapter 1: Care and Feeding of iptables

This chapter provides an introduction to packet filtering with iptables, including kernel build specifics and iptables administration. A default policy and network diagram is provided in this chapter and is referenced throughout the book. The Linux machine that runs the default policy functions as the firewall for a local area network (LAN), and attacks against this system are illustrated in later chapters.

Chapter 2: Network Layer Attacks and Defense

This chapter shows the types of attacks that exist in the network layer and what you can do about them. I'll introduce you to the iptables log- ging format and emphasize the network layer information that you can glean from iptables logs.

Chapter 3: Transport Layer Attacks and Defense

The transport layer is the realm of server reconnaissance with port scans and sweeps, and this chapter examines the inner workings of these methods. The iptables logging format is well suited to representing transport layer header information, and this is useful for detecting all sorts of mischief.

Chapter 4: Application Layer Attacks and Defense

The majority of today's attacks take advantage of the increasing complexity of applications that ride on top of the TCP/IP suite. This chapter illustrates classes of application layer attacks that iptables can be made to detect, and it introduces you to the iptables string match extension.

Chapter 5: Introducing psad: The Port Scan Attack Detector

This chapter discusses installation and configuration of psad, and shows you why it is important to listen to the stories that iptables logs have to tell.

Chapter 6: psad Operations: Detecting Suspicious Traffic

There are many features offered by psad, and these features are designed to maximize your use of iptables log messages. From port scans to probes for backdoors, psad detects and reports suspicious activity with verbose email and syslog alerts.

Chapter 7: Advanced psad Topics: From Signature Matching to OS Fingerprinting This chapter introduces you to advanced psad functionality, including integrated passive OS fingerprinting, Snort signature detection via packet headers, verbose status information, and DShield reporting. This chapter is all about showing how far iptables log information can go toward providing security data.

Chapter 8: Active Response with psad

No treatment of intrusion detection would be complete without a discus- sion of options for automatically responding to attacks. The response capabilities offered by psad are built on top of a clean interface that makes it easy to integrate with third-party software, and an example of integrating with the Swatch project is included.

Chapter 9: Translating Snort Rules into iptables Rules

The Snort IDS has shown the community the way to detect network- based attacks, and so it is logical to leverage the Snort signature lan- guage in iptables. Because iptables offers a rich logging format and the ability to inspect application layer data, a significant percentage of Snort signatures can be translated into iptables rules.

Chapter 10: Deploying fwsnort

The tedious task of translating Snort signatures into iptables rules has been automated by the fwsnort project, and this chapter shows you how it is done. Deploying fwsnort endows your iptables policy with true intru- sion detection abilities.

Chapter 11: Combining psad and fwsnort

Log messages that are generated by fwsnort are picked up and analyzed by psad for better reporting via email (integrated whois and reverse DNS lookups as well as passive OS fingerprinting are illustrated). This chapter represents the culmination of the attack detection and mitiga- tion strategies that are possible with iptables.

Chapter 12: Port Knocking vs. Single Packet Authorization

Passive authorization is becoming increasingly important for keeping networked services secure. The damaging scope of zero-day vulnerabili- ties can be severely limited by using such a technology, but not all passive authorization paradigms are robust enough for critical deployments. This chapter compares and contrasts two passive authorization mecha- nisms: port knocking and Single Packet Authorization (SPA).

Chapter 13: Introducing fwknop

There are only a few SPA implementations available today, and fwknop is one of the most actively developed and supported. This chapter shows you how to install and make use of fwknop together with iptables to main- tain a default-drop stance against all unauthenticated and unauthorized attempts to connect to your SSH daemon.

Chapter 14: Visualizing iptables Logs

The last chapter in the book wraps up with some graphical representa- tions of iptables log data. A picture can quickly illustrate trends in net- work communications that may indicate a system compromise, and by combining psad with the AfterGlow project you can see what iptables has to show you.

Appendix A: Attack Spoofing

It's exceedingly easy to parse the Snort signature ruleset, craft matching packet data, and blast it on the wire from spoofed source addresses. Appendix A discusses a sample Perl script (bundled with fwsnort) that does just this.

Appendix B: A Complete fwsnort Script

The fwsnort project creates a shell script that automates the execution of the iptables commands necessary to create an iptables policy that is capable of detecting application layer attacks. Appendix B contains a complete example of an fwsnort.sh script generated by fwsnort.

This book takes a highly applied approach. Concepts are better under- stood with real examples, and getting down into the guts of the source code or carefully examining packet traces are always excellent ways to understand what a computer is doing. It is my hope that after reading this book you will be armed with a strong working knowledge of how network attacks are detected and dealt with via iptables. Once again, I strongly encourage you to ask questions, and you can always reach me at mbr@cipherdyne.org.

1

CARE AND FEEDING OF IPTABLES

In this chapter we'll explore essential aspects of properly installing, maintaining, and interacting with the iptables firewall on Linux systems. We'll cover iptables administration from the perspectives of both kernel and userland, as well as how to build and maintain an iptables firewall policy. A default policy will be constructed that will serve as a guide throughout several chapters in the book; a script that implements it and a network diagram are included for reference in this chapter. Many of the example attacks through- out this book will be launched from hosts shown in this network diagram. Finally, we'll cover testing the default iptables policy to ensure that it is functioning as designed.

iptables

The iptables firewall is developed by the Netfilter Project (http://www .netfilter.org) and has been available to the masses as part of Linux since the release of the Linux 2.4 kernel in January 2001.

Over the years, iptables has matured into a formidable firewall with most of the functionality typically

found in proprietary commercial firewalls. For example, iptables offers comprehensive protocol state tracking, packet application layer inspection, rate limiting, and a powerful mechanism to specify a filtering policy. All major Linux distributions include iptables, and many prompt the user to deploy an iptables policy right from the installer.

The differences between the terms *iptables* and *Netfilter* have been a source of some confusion in the Linux community. The official project name for all of the packet filtering and mangling facilities provided by Linux is *Netfilter*, but this term also refers to a framework within the Linux kernel that can be used to hook functions into the networking stack at various stages. On the other hand, *iptables* uses the Netfilter framework to hook functions designed to perform operations on packets (such as filtering) into the networking stack. You can think of Netfilter as providing the framework on which iptables builds firewall functionality.

The term *iptables* also refers to the userland tool that parses the command line and communicates a firewall policy to the kernel. Terms such as *tables*, *chains*, *matches*, and *targets* (defined later in this chapter) make sense in the context of iptables.

Netfilter does not filter traffic itself—it just allows functions that *can* filter traffic to be hooked into the right spot within the kernel. (I will not belabor this point; much of the material in this book centers around iptables and how it can take action against packets that match certain criteria.) The Netfilter Project also provides several pieces of infrastructure in the kernel, such as connection tracking and logging; any iptables policy can use these facilities to perform specialized packet processing.

NOTE *In this book I will refer to log messages generated by the Netfilter logging subsystem as*

iptables log messages*; after all, packets are only logged upon matching a* `LOG` *rule that is constructed by iptables in the first place. So as to not confuse things, I will use the term* iptables *by default unless there is a compelling reason to use* Netfilter *(such as when discussing kernel compilation options or connection-tracking capabilities). Most people associate Linux firewalls with iptables, anyway.*

Packet Filtering with iptables

The iptables firewall allows the user to instrument a high degree of control over IP packets that interact with a Linux system; that control is implemented within the Linux kernel. A policy can be constructed with iptables that acts as a vigorous traffic cop—packets that are not permitted to pass fall into obliv- ion and are never heard from again, whereas packets that pass muster are sent on their merry way or altered so that they conform to local network requirements.

An iptables policy is built from an ordered set of *rules*, which describe to the kernel the actions that should be taken against certain classes of packets. Each iptables rule is applied to a chain within a table. An iptables *chain* is a

collection of rules that are compared, in order, against packets that share a common characteristic (such as being routed to the Linux system, as opposed to away from it).

Tables

A *table* is an iptables construct that delineates broad categories of function- ality, such as packet filtering or Network Address Translation (NAT). There are four tables: `filter`, `nat`, `mangle`, and `raw`. Filtering rules are applied to the `filter` table, NAT rules are applied to the `nat` table, specialized rules that alter packet data are applied to the `mangle` table, and rules that should function independently of the Netfilter connection-tracking subsystem are applied to the `raw` table.

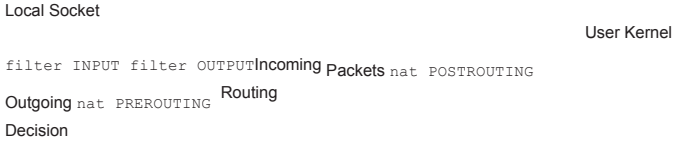
Chains

Each table has its own set of built-in chains, but user-defined chains can also be created so that the user can build a set of rules that is related by a common tag such as `INPUT_ESTABLISHED` or `DMZ_NETWORK`. The most important built-in chains for our purposes are the `INPUT`, `OUTPUT`, and `FORWARD` chains in the `filter` table:

The `INPUT` chain is traversed by packets that are destined for the local Linux system after a routing calculation is made within the kernel (i.e., packets destined for a local socket). The `OUTPUT` chain is reserved for packets that are generated by the Linux system itself. The `FORWARD` chain governs packets that are routed through the Linux system (i.e., when the iptables firewall is used to connect one network to another and packets between the two networks must flow through the firewall).

Two additional chains that are important for any serious iptables deploy- ment are the `PREROUTING` and `POSTROUTING` chains in the `nat` table, which are used to modify packet headers before and after an IP routing calculation is made within the kernel. Sample iptables commands illustrate the usage of the `PREROUTING` and `POSTROUTING` chains later in this chapter, but in the mean- time, Figure 1-1 shows how packets flow through the `nat` and `filter` tables within the kernel.

Figure 1-1: iptables packet flow



Match

es

Every iptables rule has a set of matches along with a *target* that tells iptables what to do with a packet that conforms to the rule. An iptables *match* is a condition that must be met by a packet in order for iptables to process the packet according to the action specified by the rule target. For example, to apply a rule only to TCP packets, you can use the `--protocol` match.

Each match is specified on the iptables command line. The most import- ant iptables matches for this book are listed below. (You'll see more about matches in "Default iptables Policy" on page 20 when we discuss the default iptables policy used throughout this book.)

`--source (-s)` Match on a source IP address or network **`--destination (-d)`** Match on a destination IP address or network **`--protocol (-p)`** Match on an IP value **`--in-interface (-i)`** Input interface (e.g., eth0) **`--out-interface (-o)`** Output interface **`--state`** Match on a set of connection states **`--string`** Match on a sequence of application layer data bytes **`--comment`** Associate up to 256 bytes of comment data with a rule within kernel memory

Targ ets

Finally, iptables supports a set of targets that trigger an action when a packet matches a rule.[1] The most important targets used in this book are as follows:

`ACCEPT` Allows a packet to continue on its way. **`DROP`** Drops a packet. No further processing is performed, and as far as the receiving stack is concerned, it is as though the packet was never sent. **`LOG`** Logs a packet to syslog. **`REJECT`** Drops a packet and simultaneously sends an appropriate response packet (e.g., a TCP Reset packet for a TCP connection or an ICMP Port Unreachable message for a UDP packet). **`RETURN`** Continues processing a packet within the calling chain.

We'll build ample iptables rules that use several of the matches and targets discussed above in "Default iptables Policy" on page 20.

Installing iptables

Because iptables is split into two fundamental components (kernel modules and the userland administration program), installing iptables involves com- piling and installing both the Linux kernel and the userland binary. The

[1] Note that *matching* here is used to mean that a packet conforms to all of the match criteria contained within an iptables rule.

kernel source code contains many Netfilter subsystems, and the essential packet-filtering capability is enabled by default in the pristine authoritative kernels released on the official Linux Kernel Archives website, http://www.kernel.org.

In some of the earlier 2.6 kernels (and all of the 2.4 kernels), the Netfilter compilation options were not enabled by default. However, because the soft- ware provided by the Netfilter Project has achieved a high level of quality over the years, the kernel maintainers felt it had reached a point where using iptables on Linux should not require you to recompile the kernel. Recent kernels allow you to filter packets by default with an iptables policy.

While many Linux distributions come with pre-built kernels that already have iptables compiled in, the default kernel configuration in a kernel down- loaded from http://www.kernel.org tries to stay as lean and mean as possible out of the box, so not all Netfilter subsystems may be enabled. For example, the Netfilter connection-tracking capability is not enabled by default in the 2.6.20.1 kernel (the most recent kernel version as of this writing). Hence, it is important to understand the process of recompiling the kernel so that iptables policies can make use of additional functionality.

NOTE *Throughout this chapter, some of the compilation output and installation commands have been abbreviated to save space and keep the focus on what is important.*

The most important step towards building a Linux system that can func- tion as an iptables firewall is the proper configuration and compilation of the Linux kernel. All heavy network-processing and comparison functions in iptables take place within the kernel, and we'll begin by compiling the latest version of the kernel from the 2.6 stable series. Although a complete treatment of the vagaries of the kernel compilation process is beyond the scope of this book, we'll discuss enough of the process for you to compile in and enable the critical capabilities of packet filtering, connection tracking, and logging. As far as other kernel compilation options not related to Netfilter subsystems, such as processor architecture, network interface driver(s), and filesystem support, I'll assume that you've chosen the correct options such that the resulting kernel will function correctly on the hardware on which it is deployed.

NOTE *For more information on compiling the 2.6 series kernel, see the Kernel Rebuild Guide written by Kwan Lowe (http://www.digitalhermit.com/~kwan/kernel.html). For the older 2.4 kernels, see the Kernel-HOWTO written by Brian Ward (http://www.tldp.org/HOWTO/Kernel-HOWTO.html), or refer to any good book on Linux system administration. Brian Ward's* How Linux Works *(No Starch Press, 2004) also covers kernel compilation.*

Before you can install the Linux kernel, you need to download and unpack it. The following commands accomplish this for the 2.6.20.1 kernel. (In these commands, I assume the directory

/usr/src is writable by the current user.)

NOTE *Except where otherwise noted, this chapter is written from the perspective of the 2.6-series kernel because it represents the latest and greatest progeny of the Linux kernel developers. In general, however, the same strategies also apply to the 2.4-series kernel.*

```
$ /usr/src $ wget
http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.20.1.tar
.bz2 $ tar xfj linux-2.6.20.1.tar.bz2 $ ls –ld linux-2.6.20.1
drwxr-xr-x 18 mbr users 600 Jun 16 20:48 linux-2.6.20.1
```

Although I have chosen specific kernel versions in the commands above, the analogous commands apply for newer kernel versions. For example when, say, the 2.6.20.2 kernel is released, you only need to substitute `2.6.20.1` with **`2.6.20.2`** in the above commands.

NOTE *One thing to keep in mind is that the load on the kernel.org webserver has been steadily increasing over the years, and a random glance at the bandwidth utilization graphs on http://www.kernel.org shows the current utilization at well over 300 Mbps. To help reduce the load, the kernel can be downloaded from one of the mirrors listed at http:// www.kernel.org/mirrors. Once you have a particular version of the kernel sources on your system, you can download and apply a kernel patch file to upgrade to the next version. (The patch files are much smaller than the kernel itself.)*

Kernel Configuration

Before you can begin compiling, you must construct a kernel configuration file. Fortunately, the process of building this file has been automated by kernel developers, and it can be initiated with a single command (within the /usr/src/linux-2.6.20.1 directory):

```
$ make
menuconfig
```

The `make menuconfig` command launches the Ncurses interface in which you can select various compile options. (You can call the X Windows or terminal interface with the commands `make xconfig` and `make config`, respect- ively.) I've chosen the Ncurses interface because it provides a nice balance between the spartan terminal interface and the relatively expensive X Windows interface. The Ncurses interface also easily lends itself to the configuration of a remote Linux kernel across an SSH session without having to forward an X Windows connection.

After executing `make menuconfig`, we are presented with several config- uration sections ranging from Code Maturity Level options to Library Rou- tines. Most Netfilter compilation

options for the 2.6-series kernel are located within a section called Network Packet Filtering Framework (Netfilter) under Networking Networking Options.

Essential Netfilter Compilation Options

Some of the more important options to enable within the kernel configura- tion file include Netfilter connection tracking, logging, and packet filtering. (Recall that iptables builds a policy by using the in-kernel framework pro- vided by Netfilter.)
There are two additional configuration sections in the Network Packet Filtering Framework (Netfilter) section—Core Netfilter Configuration and IP: Netfilter Configuration.

Core Netfilter Configuration The Core Netfilter Configuration section contains several important options that should all be enabled:

Comment match support FTP support
Length match support Limit match support
MAC address match support `MARK` target
support Netfilter connection tracking
support Netfilter `LOG` over NFNETLINK
interface Netfilter netlink interface Netfilter
Xtables support State match support String
match support

IP: Netfilter Configuration With the Core Netfilter Configuration section completed, we'll move on to the IP: Netfilter Configuration section. The options that should be enabled within this section are as follows:

ECN target support Full NAT IP address range match
support IP tables support (required for
filtering/masq/NAT) IPv4 connection tracking support
(required for NAT) `LOG` target support `MASQUERADE` target
support Owner match support Packet filtering Packet
mangling

`raw` table support (required for NOTRACK/TRACE)

Recent match support `REJECT` target support `TOS` match support `TOS` target support `TTL` match support `TTL` target support `ULOG` target support

In the 2.6 kernel series, the individual compilation sections underwent a major reorganization. In the older 2.4 series, the IP: Netfilter Configuration section can be found underneath Networking Options, and this section is only visible if the Network Packet Filtering option is enabled.

Finishing the Kernel Configuration

Having configured the 2.6.20.1 kernel with the required Netfilter support via the `menuconfig` interface, save the kernel configuration file by selecting **Exit** until you see the message *Do you wish to save your new kernel configuration?* Answer **Yes**.

After saving the new kernel configuration, you are dropped back to the command shell where you can examine the resulting Netfilter compilation options via the following commands.

NOTE *The output of these commands is too long to include here, but most Netfilter options, such as `CONFIG_IP_NF_NAT` and `CONFIG_NETFILTER_XT_MATCH_STRING`, for example, contain either the substring `_NF_` or the substring `NETFILTER`.*

```
$ grep "_NF_"
.config $ grep
NETFILTER .config
```

Loadable Kernel Modules vs. Built-in Compilation and Security

Most of the Netfilter subsystems enabled in the previous section may be com- piled either as a Loadable Kernel Module (LKM), which can be dynamically loaded or unloaded into or out of the kernel at run time, or compiled directly into the kernel, in which case they cannot be loaded or unloaded at run time. In the configuration section above, we have chosen to compile most Netfilter subsystems as LKMs.

There is a security trade-off between compiling functionality as an LKM and compiling directly into the kernel. On one hand, any feature that is compiled as an LKM can be removed from a running kernel with the `rmmod` command. This can provide an advantage if a security vulnerability is discov- ered within the module, because in some cases the vulnerability can be miti- gated just by unloading the module. Too, if the vulnerability has been patched in the kernel sources, the module can be recompiled and redeployed without ever taking the system down completely; fixing the vulnerability would involve zero downtime.

NOTE *Netfilter subsystems in the kernel are not immune from the occasional security*

vulnera-

bility. For example, a vulnerability was discovered in the code that handles TCP options in the Netfilter logging subsystem (see http://www.netfilter.org/security/2004-06-30- 2.6-tcpoption.html). If the logging subsystem was compiled as a module, the kernel can be protected by sacrificing the ability of iptables to create log messages by unloading the module, which seems like a good trade-off.

On the other hand, if a vulnerability is discovered within the code that implements a feature and this code is compiled directly into the kernel, the only way to fix the vulnerability is to apply a patch, recompile, and then reboot the entire system into the new (fixed) kernel. For mission-critical systems (such as a corporate DNS server), this may not be feasible until an outage window can be scheduled, and in the meantime the system may be vulner- able to a kernel-level compromise.

ROOTKIT THREAT

The story does not end here, however. Compiling a kernel with loadable module support opens up a sinister possibility: If an attacker successfully compromises the system, having module support in the kernel makes it easier for the attacker to install a kernel-level rootkit. Once the kernel itself is compromised, all sorts of mischief can be levied against the system.

Compromising the kernel itself represents the crown jewel of all compromises; filesystem integrity checkers such as Tripwire can be fooled, processes can be hid- den, and network connections can be shielded from the view of tools like netstat and lsof, and even from packet sniffers (executed locally). Simply compiling the kernel without module support is not a foolproof solution, however, since not all kernel-level rootkits require the host kernel to offer module support. For example, the SucKIT rootkit can load itself into a running kernel by directly manipulating kernel memory through the /dev/kmem character device.* The SucKIT rootkit was introduced to the security community in the *Phrack* magazine article “Linux on-the-fly kernel patching without LKM” (see http://www.phrack.org).

*A *character device* is an interface to the kernel that can be accessed as a stream of bytes instead of just by discrete block sizes, as in the case of a block device. Examples of character devices include /dev/console and the serial port device files, such as /dev/ttyS0.

The power of module loading and unloading provides a degree of flexibility that is attractive, so this is the strategy I chose here. When making your own choice, be sure to consider the trade-offs.

Security and Minimal Compilation

Regardless of the strategy you choose for compiling Netfilter subsystems—whether as LKM’s or directly into the kernel—an overriding fact in computer security is that complexity breeds insecurity; more complex systems are harder to secure. Fortunately, iptables is highly configurable both in terms of the run-time rules language used to describe how to process and filter network traffic and also in

terms of the categories of supported features controlled by the kernel compilation options.

To reduce the complexity of the code running in the kernel, do not compile features that you don’t need.

Removing unnecessary code from a running kernel helps to minimize the risks from as yet undiscovered vulner- abilities lurking in the code.

For example, if you have no need for logging support, simply do not enable the Log Target Support option in the menuconfig interface. If you have no need for the stateful tracking of FTP connections, leave the FTP Protocol Support option disabled. If you do not need to be able to write filter rules against MAC addresses in Ethernet headers, disable the MAC Address Match Support option.

Only compile in the features that are absolutely necessary to meet the networking and security needs of the local network and/or host.

Kernel Compilation and Installation

Now that our kernel is configured, we'll move on to the compilation and installation. As previously mentioned, we assume that all other necessary kernel options (such as processor architecture) have been selected for the proper support of the hardware on which the new kernel will run.

To compile and install the new 2.6.20.1 kernel within the /boot partition, execute the following commands:

```
$ make $ su - Password: # mount
/boot # cd
/usr/src/linux-2.6.20.1 # make
install && make modules_install
```

The successful conclusion of the above commands heralds the need to configure the bootloader and finally to boot into the new 2.6.20.1 kernel. Assuming that you're using the GRUB bootloader and that the mount point for the root partition is /dev/hda2, add the following lines to the /boot/grub/grub.conf file using your favorite editor:

```
title linux-2.6.20.1 root (hd0,0)
kernel /boot/vmlinuz-2.6.20.1
root=/dev/hda2
```

Now, reboot!

```
# shutdown -r
now
```

Installing the iptables Userland Binaries

Having installed and booted into a kernel that has Netfilter hooks compiled in, we'll now install the latest version of the iptables userland program. To do so, first download and unpack the latest iptables sources in the /usr/local/src directory, and then check the MD5 sum[2] against the published value at http://www .netfilter.org:

```
$ cd /usr/local/src/ $ wget
http://www.netfilter.org/projects/iptables/files/iptables-1.3.7.ta
r.bz2 $ md5sum 1.3.7.tar.bz2 dd965bdacbb86ce2a6498829fddda6b7
iptables-1.3.7.tar.bz2 $ tar xfj iptables-1.3.7.tar.bz2 $ cd
iptables-1.3.7
```

For the compilation and installation steps of the iptables binary, recall that we compiled the kernel within the directory /usr/src/linux-2.6.20.1; com- piling iptables requires access to the kernel source code because it compiles against C header files in directories such as include/linux/netfilter_ipv4 in the kernel source tree. We'll use the /usr/src/linux-2.6.20.1 directory to define the `KERNEL_DIR` variable on the command line, and the `BINDIR` and `LIBDIR` variables allow us to control the paths where the iptables binary and libraries are installed. You can compile and install iptables as follows:

```
$ make KERNEL_DIR=/usr/src/linux-2.6.20.1 BINDIR=/sbin
LIBDIR=/lib $ su - Password: # cd /usr/local/src/iptables-1.3.7
# make install KERNEL_DIR=/usr/src/linux-2.6.20.1 BINDIR=/sbin
LIBDIR=/lib
```

For the final proof that we have installed iptables and that it can interact with the running 2.6.20.1 kernel, we'll issue commands to display the iptables version number and then instruct it to list the current ruleset in the `INPUT`, `OUTPUT`, and `FORWARD` chains (which at this point contain no active rules):

```
# which iptables /sbin/iptables # iptables
-V iptables v1.3.7 # iptables -nL 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
```

[2] You should also check the digital signature made with GnuPG against the published value at http://www.netfilter.org. This requires importing the Netfilter GnuPG public key, and running the `gpg --verify` command against the signature file. Details of this process for the psad project can be found in Chapter 5, and similar steps apply here to the iptables-1.3.7 tarball.

NOTE *Most Linux distributions already have iptables installed, so you may not need to go through the installation process above. However, to ensure you have a system that is prepared for the discussion in this book, it may be a good idea to have the latest version of*

iptables installed. As you will see in Chapter 9, the string matching capability is crit- ical for running fwsnort, so you may need to upgrade your kernel if it doesn't already support this (see "Kernel Configuration" on page 14).

Default iptables Policy

We now have a functioning Linux system with iptables installed. The remainder of this chapter will concentrate on various administrative and run-time aspects of iptables firewalls.

We'll begin by constructing a Bourne shell script (iptables.sh) to imple- ment an iptables filtering policy tailored for a modest network with a perm- anent Internet connection. This policy will be used throughout the rest of the book and serves as a common ground—we will refer to this policy in several subsequent chapters. You can also download the iptables.sh script from http://www.cipherdyne.org/LinuxFirewalls. But first, here is some background information on iptables.

Policy Requirements

Let's define the requirements for an effective firewall configuration for a network consisting of several client machines and two servers. The servers (a webserver and a DNS server) must be accessible from the external network. Systems on the internal network should be allowed to initiate the following types of traffic through the firewall to external servers:

Domain Name System (DNS) queries File Transfer Protocol (FTP) transfers Network Time Protocol (NTP) queries Secure SHell (SSH) sessions Simple Mail Transfer Protocol (SMTP) sessions Web sessions over HTTP/HTTPS whois queries

Except for access to the services listed above, all other traffic should be blocked. Sessions initiated from the internal network or directly from the firewall should be statefully tracked by iptables (with packets that do not conform to a valid state logged and dropped as early as possible), and NAT services should also be provided.

In addition, the firewall should also implement controls against spoofed packets from the internal network being forwarded to any external IP address:

The firewall itself must be accessible via SSH from the internal network, but from nowhere else unless it is running fwknop for authentication

(covered in Chapter 13); SSH should be the only server process running on the firewall. The firewall should accept ICMP Echo Requests from both the internal and external networks, but unsolicited ICMP packets that are not Echo Requests should be dropped from any source IP address. Lastly, the firewall should be configured with a default *log and drop stance* so that any stray packets, port scans, or other connection attempts that are not explicitly allowed

through will be logged and dropped.

NOTE *We'll assume that the external IP address on the firewall is statically assigned by the ISP, but a dynamically assigned IP address would also work because we restrict packets on the external network by interface name on the firewall instead of by IP address.*

To simplify the task of building the iptables policy, assume there is a single internal network with a non-routable network address of 192.168.10.0[3] and a Class C subnet mask 255.255.255.0 (or /24 in CIDR notation).

The internal network interface on the firewall (see Figure 1-2) is eth1 with IP address 192.168.10.1, and all internal hosts have this address as their default gateway. This allows internal systems to route all packets destined for systems that are not within the 192.168.10.0/24 subnet out through the fire- wall. The external interface on the firewall is eth0, and so as to remain network agnostic, we designate an external IP address of 71.157.*X.X* to this interface.

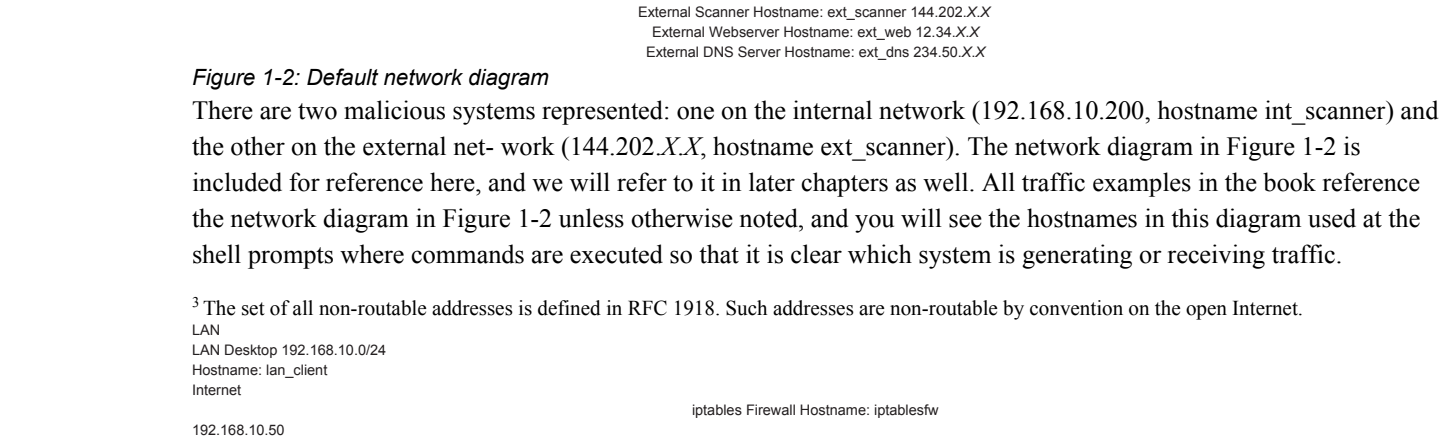


Figure 1-2: Default network diagram

There are two malicious systems represented: one on the internal network (192.168.10.200, hostname int_scanner) and the other on the external net- work (144.202.*X.X*, hostname ext_scanner). The network diagram in Figure 1-2 is included for reference here, and we will refer to it in later chapters as well. All traffic examples in the book reference the network diagram in Figure 1-2 unless otherwise noted, and you will see the hostnames in this diagram used at the shell prompts where commands are executed so that it is clear which system is generating or receiving traffic.

[3] The set of all non-routable addresses is defined in RFC 1918. Such addresses are non-routable by convention on the open Internet.

iptables.sh Script Preamble

To begin the iptables.sh script, it is useful to define three variables, `IPTABLES` and `MODPROBE` (for the paths to the iptables and modprobe binaries) and `INT_NET` (for the internal subnet address and mask), that will be used throughout the script (see below). At any existing iptables rules are removed from the running kernel, and the filtering policy is set to `DROP` on the `INPUT`, `OUTPUT`, and `FORWARD` chains. Also, the connection-tracking modules are loaded with the `modprobe` command.

```
[iptablesfw]# cat
iptables.sh #!/bin/sh
IPTABLES=/sbin/iptables
MODPROBE=/sbin/modprobe
INT_NET=192.168.10.0/24

### flush existing rules and set chain policy
setting to DROP echo "[+] Flushing existing iptables
rules..." $IPTABLES -F $IPTABLES -F -t nat $IPTABLES
-X $IPTABLES -P INPUT DROP $IPTABLES -P OUTPUT DROP
$IPTABLES -P FORWARD DROP ### load
connection-tracking modules $MODPROBE ip_conntrack
$MODPROBE iptable_nat $MODPROBE ip_conntrack_ftp
$MODPROBE ip_nat_ftp
```

The INPUT Chain

The `INPUT` chain is the iptables construct that governs whether packets that are destined for the local system (that is, after the result of a routing calcu- lation made by the kernel IP stack designates that the packet is destined for a local IP address) may talk to a local socket. If the first rule in the `INPUT` chain instructs iptables to drop all packets (or if the policy setting of the `INPUT` chain is set to `DROP`), then all efforts to communicate directly with the system over any IP communications (such as TCP, UDP, or ICMP) will fail. The *Address Resolution Protocol (ARP)* is also an important class of traffic that is ubiquitous on Ethernet networks. However, because ARP works at the data link layer instead of the network layer, iptables cannot filter such traffic, since it only filters IP traffic and overlying protocols. Hence, ARP requests and replies are sent and received regardless of the iptables policy. (It is possible to filter ARP traffic with arptables, but a discus- sion of this topic is beyond the scope of this book, since we generally con- centrate on the network layer and above.)

NOTE *iptables can filter IP packets based on data link layer MAC addresses, but only if the kernel is compiled with the MAC address extension enabled. In the 2.4 kernel series, the MAC address extension must be manually enabled, but the 2.6 kernel series enables it by default.*

Continuing with the development of the iptables shell script, after the preamble, we use the following

commands to set up the `INPUT` chain.

```
                    ###### INPUT chain ###### echo "[+] Setting up INPUT chain..." ###
                    state tracking rules $IPTABLES -A INPUT -m state --state INVALID
                    -j LOG --log-prefix "DROP INVALID " --log-ip-options
                    --log-tcp-options $IPTABLES -A INPUT -m state --state INVALID -j
                    DROP $IPTABLES -A INPUT -m state --state ESTABLISHED,RELATED -j
                    ACCEPT

### anti-spoofing rules $IPTABLES -A INPUT -i eth1 -s ! $INT_NET -j LOG --log-prefix
                    "SPOOFED PKT " $IPTABLES -A INPUT -i eth1 -s ! $INT_NET -j DROP

### ACCEPT rules $IPTABLES -A INPUT -i eth1 -p tcp -s $INT_NET --dport 22 --syn
                    -m state --state NEW -j ACCEPT $IPTABLES -A INPUT -p icmp
                    --icmp-type echo-request -j ACCEPT

### default INPUT LOG rule $IPTABLES -A INPUT -i ! lo -j LOG --log-prefix "DROP "
                    --log-ip-options --log-tcp-options
```

Recall that our firewall policy requirements mandate that iptables statefully tracks connections; packets that do not match a valid state should be logged and dropped early. This is accomplished by the three iptables com- mands beginning at above; you will see a similar set of three commands for the `OUTPUT` and `FORWARD` chains as well. The state match is used by each of these rules, along with the criteria of `INVALID`, `ESTABLISHED`, or `RELATED`. The `INVALID` state applies to packets that cannot be identified as belonging to any existing connection—for example, a TCP FIN packet that arrives out of the blue (i.e., when it is not part of any TCP session) would match the `INVALID` state. The `ESTABLISHED` state triggers on packets only after the Netfilter connection- tracking subsystem has seen packets in both directions (such as acknowledg- ment packets in a TCP connection through which data is being exchanged). The `RELATED` state describes packets that are starting a new connection[4] in the Netfilter connection-tracking subsystem, but this connection is associated with an existing one—for example, an ICMP Port Unreachable message that is returned after a packet is sent to a UDP socket where no server is bound. Next, anti-spoofing rules are added at so packets that originate from the internal network must have a source address within the 192.168.10.0/24 subnet. At are two `ACCEPT` rules for SSH connections from the internal network, and ICMP Echo Requests are accepted from any source. The rule that accepts SSH connections uses the state match with a state of `NEW` together with the iptables `--syn` command-line argument. This only matches on TCP packets with FIN, RST, and ACK flags zeroed-out and the SYN flag set, and then only if the `NEW` state is matched (which means that the packet is starting a new connection, as far as the connection-tracking subsystem is concerned).

[4] Here *connection* is the tracking mechanism that Netfilter uses to categorize packets.

Finally at is the default `LOG` rule.[5] Recall from the script preamble that packets that are not accepted by some rule within the `INPUT` chain will be dropped by the `DROP` policy assigned to the chain; this also applies to the `OUTPUT` and `FORWARD` chains. As you can see, the configuration of the `INPUT` chain is exceedingly easy, since we only need to accept incoming connection requests to the SSH daemon from the internal network, enable state tracking for locally generated network traffic, and finally log and drop unwanted packets (includ- ing spoofed packets from the internal network). Similar configurations apply to `OUTPUT` and `FORWARD` chains, as you'll see below.

The OUTPUT Chain

The `OUTPUT` chain allows iptables to apply kernel-level controls to network packets generated by the local system. For example, if an SSH session is initiated to an external system by a local user, the `OUTPUT` chain could be used to either permit or deny the outbound SYN packet.

The commands in the iptables.sh script that build the `OUTPUT` chain ruleset appear below:

```
###### OUTPUT chain ###### echo "[+] Setting up OUTPUT
chain..." ### state tracking rules $IPTABLES -A OUTPUT -m
state --state INVALID -j LOG --log-prefix "DROP INVALID "
--log-ip-options --log-tcp-options $IPTABLES -A OUTPUT -m
state --state INVALID -j DROP $IPTABLES -A OUTPUT -m state
--state ESTABLISHED,RELATED -j ACCEPT

### ACCEPT rules for allowing connections out $IPTABLES -A OUTPUT -p tcp --dport 21
--syn -m state --state NEW -j ACCEPT $IPTABLES -A OUTPUT -p tcp
--dport 22 --syn -m state --state NEW -j ACCEPT $IPTABLES -A
OUTPUT -p tcp --dport 25 --syn -m state --state NEW -j ACCEPT
$IPTABLES -A OUTPUT -p tcp --dport 43 --syn -m state --state NEW
-j ACCEPT $IPTABLES -A OUTPUT -p tcp --dport 80 --syn -m state
--state NEW -j ACCEPT $IPTABLES -A OUTPUT -p tcp --dport 443
--syn -m state --state NEW -j ACCEPT $IPTABLES -A OUTPUT -p tcp
--dport 4321 --syn -m state --state NEW -j ACCEPT $IPTABLES -A
OUTPUT -p udp --dport 53 -m state --state NEW -j ACCEPT
$IPTABLES -A OUTPUT -p tcp --dport 53 -m state --state NEW -j
ACCEPT $IPTABLES -A OUTPUT -p icmp --icmp-type echo-request -j
ACCEPT

### default OUTPUT LOG rule $IPTABLES -A OUTPUT -o ! lo -j
LOG --log-prefix "DROP " --log-ip-options --log-tcp-options
```

In accordance with our policy requirements, at we'll assume that connections initiated from the firewall itself will be to download patches or software over FTP, HTTP, or HTTPS; to initiate outbound SSH and SMTP connections; or to issue DNS or whois queries against other systems.

[5] One thing to note about the iptables.sh script is that all of the `LOG` rules are built with the `--log-ip-options` and `--log-tcp-options` command-line arguments. This allows the resulting iptables syslog messages to include the IP and TCP options portions of the IP and TCP headers if the

packet that matches the `LOG` rule contains them. This functionality is important for both attack detection and passive OS fingerprinting operations performed by psad (see Chapter 7).

The FORWARD Chain

So far the rules we have added to the iptables filtering policy strictly govern the ability of packets to interact directly with the firewall system. Such packets are either destined for or emanate from the firewall operating system and include packets such as connection requests to the SSH daemon from internal systems or locally initiated connections to external sites to download security patches.

Now let's look at the iptables rules that pertain to packets that do not have a source or destination address associated with the firewall, but which nevertheless attempt to route through the firewall system. The iptables `FORWARD` chain in the `filter` table provides the ability to wrap access controls around packets that are forwarded across the firewall interfaces:

```
###### FORWARD chain ###### echo "[+] Setting up FORWARD
chain..." ### state tracking rules $IPTABLES -A FORWARD -m
state --state INVALID -j LOG --log-prefix "DROP INVALID "
--log-ip-options --log-tcp-options $IPTABLES -A FORWARD -m
state --state INVALID -j DROP $IPTABLES -A FORWARD -m state
--state ESTABLISHED,RELATED -j ACCEPT

### anti-spoofing rules $IPTABLES -A FORWARD -i eth1 -s ! $INT_NET
-j LOG --log-prefix "SPOOFED PKT " $IPTABLES -A FORWARD -i eth1 -s
! $INT_NET -j DROP

### ACCEPT rules $IPTABLES -A FORWARD -p tcp -i eth1 -s $INT_NET --dport 21 --syn -m
state --state NEW -j ACCEPT $IPTABLES -A FORWARD -p tcp -i eth1
-s $INT_NET --dport 22 --syn -m state --state NEW -j ACCEPT
$IPTABLES -A FORWARD -p tcp -i eth1 -s $INT_NET --dport 25 --syn
-m state --state NEW -j ACCEPT $IPTABLES -A FORWARD -p tcp -i
eth1 -s $INT_NET --dport 43 --syn -m state --state NEW -j ACCEPT
$IPTABLES -A FORWARD -p tcp --dport 80 --syn -m state --state NEW
-j ACCEPT $IPTABLES -A FORWARD -p tcp --dport 443 --syn -m state
--state NEW -j ACCEPT $IPTABLES -A FORWARD -p tcp -i eth1 -s
$INT_NET --dport 4321 --syn -m state --state NEW -j ACCEPT
$IPTABLES -A FORWARD -p udp --dport 53 -m state --state NEW -j
ACCEPT $IPTABLES -A FORWARD -p tcp --dport 53 -m state --state
NEW -j ACCEPT $IPTABLES -A FORWARD -p icmp --icmp-type
echo-request -j ACCEPT

### default log rule $IPTABLES -A FORWARD -i ! lo -j LOG
--log-prefix "DROP " --log-ip-options --log-tcp-options
```

Similar to the rules of the `OUTPUT` chain, at FTP, SSH, SMTP, and whois connections are allowed to be

initiated out through the firewall, except that such connections must originate from the internal subnet on the subnet- facing interface (eth1). HTTP, HTTPS, and DNS traffic is allowed through

from any source because we need to allow external addresses to interact with the

internal web- and DNS servers (after being NATed; see the following section, "Network Address Translation").

Network Address Translation

The final step in the construction of our iptables policy is to enable the translation of the non-routable 192.168.10.0/24 internal addresses into the routable external 71.157.*X.X* address. This applies to inbound connections to the web- and DNS servers from external clients, and also to outbound connections initiated from the systems on the internal network. For con- nections initiated from internal systems, we'll use the source NAT (`SNAT`) target, and for connections that are initiated from external systems, we'll use the destination NAT (`DNAT`) target.

The iptables `nat` table is dedicated to all NAT rules, and within this table there are two chains: `PREROUTING` and `POSTROUTING`. The `PREROUTING` chain is used to apply rules in the `nat` table to packets that have not yet gone through the routing algorithm in the kernel in order to determine the interface on which they should be transmitted. Packets that are processed in this chain have also not yet been compared against the `INPUT` or `FORWARD` chains in the `filter` table.

The `POSTROUTING` chain is responsible for processing packets once they have gone through the routing algorithm in the kernel and are just about to be transmitted on the calculated physical interface. Packets processed by this chain have passed the requirements of the `OUTPUT` or `FORWARD` chains in the `filter` table (as well as requirements mandated by other tables that may be registered, such as the `mangle` table).

NOTE *For a complete explanation of how iptables does NAT, see http://www.netfilter.org/documentation/HOWTO/NAT-HOWTO.html.*

```
###### NAT rules ###### echo "[+] Setting up NAT
rules..." $IPTABLES -t nat -A PREROUTING -p tcp --dport
80 -i eth0 -j DNAT --to 192.168.10.3:80 $IPTABLES -t
nat -A PREROUTING -p tcp --dport 443 -i eth0 -j DNAT
--to 192.168.10.3:443 $IPTABLES -t nat -A PREROUTING -p
tcp --dport 53 -i eth0 -j DNAT --to 192.168.10.4:53
$IPTABLES -t nat -A POSTROUTING -s $INT_NET -o eth0 -j
MASQUERADE
```

Referring to the network diagram in Figure 1-2, the IP addresses of the web- and DNS servers are 192.168.10.3 and 192.168.10.4 in the internal net- work. The iptables commands required to provide NAT functionality are displayed above (note the restriction of the commands to the `nat` table through the use of the `-t` option). The three `PREROUTING` rules at allow web services and DNS requests from the external network to be sent to the appropriate internal servers. The final `POSTROUTING` rule at allows connections that orig- inate from the internal non-routable network and destined for the external Internet to look as though they come from the IP address 71.157.*X.X*.

The very last step in building the iptables policy is to enable IP forward- ing in the Linux kernel:

```
###### forwarding ###### echo
"[+] Enabling IP forwarding..."
echo 1 >
/proc/sys/net/ipv4/ip_forward
```

Activating the Policy

One of the really nice things about iptables is that instantiating a policy within the kernel is trivially easy through the execution of iptables com- mands—there are no heavyweight user interfaces, binary file formats, or bloated management protocols (like the ones developed by some proprietary vendors of other security products). Now that we have a shell script that cap- tures the iptables commands (once again, you can download the complete script from http://www.cipherdyne.org/LinuxFirewalls), let's execute it:

```
[iptablesfw]# ./iptables.sh [+]
Flushing existing iptables
rules... [+] Setting up INPUT
chain... [+] Setting up OUTPUT
chain... [+] Setting up FORWARD
chain... [+] Setting up NAT
rules... [+] Enabling IP
forwarding...
```

iptables-save and iptables-restore

All of the previous iptables commands in the iptables.sh script are executed one at a time in order to instantiate new rules, set the default policy on a chain, or delete old rules. Each command requires a separate execution of the iptables userland binary to create the iptables policy. Hence, this is not an optimal solution for bringing the policy into existence quickly at system boot, particularly when the number of iptables rules grows into the hundreds (which can happen with a policy built by fwsnort, as we will see in Chapter 10). A much faster mechanism is provided by the commands `iptables-save` and `iptables-restore`, which are installed within the same directory (/sbin in our case) as the main iptables program. The `iptables-save` command builds a file that contains all iptables rules in a running policy in human-readable format. This format can be interpreted by the `iptables-restore` program, which takes each of the rules listed in the ipt.save file and instantiates it within a running kernel. A single execution of the `iptables-restore` program recreates an entire iptables policy in the kernel; multiple executions of the iptables program are not necessary. This makes the

`iptables-save` and `iptables-restore` commands ideal for rapid deployment of iptables rulesets, and I illustrate this process with the following two commands:

```
[iptablesfw]# iptables-save >
/root/ipt.save [iptablewfw]# cat
/root/ipt.save | iptables-restore
```

The contents of the ipt.save file are organized by iptables table, and within each section devoted to an individual table, ipt.save is further organ- ized by iptables chain. A line that begins with an asterisk (`*`) character followed by a table name (such as `filter`) denotes the beginning of a section in the ipt.save file that describes a particular table. Following this are lines that track packet and bytes counts for each chain associated with the table.

The next portion of the ipt.save file is a complete description of all iptables rules organized by chain. These lines allow the actual iptables rule- set to be reconstructed by `iptables-restore`; even including packet and byte counts for each rule if the `-c` option to `iptables-save` is used.

Lastly, the word `COMMIT` on a line by itself concludes the section of the ipt.save file that characterizes the iptables table. This line constitutes the end- ing marker for all information associated with the table. Below is a complete example of what the `filter` table section looks like once we have executed all of the iptables commands up to this point in the chapter:

```
# Generated by iptables-save v1.3.7 on Sat Apr 14 17:35:22 2007
*filter :INPUT DROP [0:0] :FORWARD DROP [0:0] :OUTPUT DROP [2:112]
-A INPUT -m state --state INVALID -j LOG --log-prefix "DROP
INVALID " --log-tcp-options --log-ip-options -A INPUT -m state
--state INVALID -j DROP -A INPUT -m state --state
RELATED,ESTABLISHED -j ACCEPT -A INPUT -s !
192.168.10.0/255.255.255.0 -i eth1 -j LOG --log-prefix "SPOOFED
PKT " -A INPUT -s ! 192.168.10.0/255.255.255.0 -i eth1 -j DROP -A
INPUT -s 192.168.10.0/255.255.255.0 -i eth1 -p tcp -m tcp --dport
22 --tcp-flags FIN,SYN,RST,ACK SYN -m state --state NEW -j ACCEPT
-A INPUT -p icmp -m icmp --icmp-type 8 -j ACCEPT -A INPUT -i ! lo
-j LOG --log-prefix "DROP " --log-tcp-options --log-ip-options -A
FORWARD -m state --state INVALID -j LOG --log-prefix "DROP INVALID
" --log-tcp-options --log-ip-options -A FORWARD -m state --state
INVALID -j DROP -A FORWARD -m state --state RELATED,ESTABLISHED -j
ACCEPT -A FORWARD -s ! 192.168.10.0/255.255.255.0 -i eth1 -j LOG
--log-prefix "SPOOFED PKT " -A FORWARD -s !
192.168.10.0/255.255.255.0 -i eth1 -j DROP -A FORWARD -s
192.168.10.0/255.255.255.0 -i eth1 -p tcp -m tcp --dport 21
--tcp-flags FIN,SYN,RST,ACK SYN -m state --state NEW -j ACCEPT -A
FORWARD -s 192.168.10.0/255.255.255.0 -i eth1 -p tcp -m tcp
--dport 22 --tcp-flags FIN,SYN,RST,ACK SYN -m state --state NEW -j
ACCEPT -A FORWARD -s 192.168.10.0/255.255.255.0 -i eth1 -p tcp -m
tcp --dport 25 --tcp-flags FIN,SYN,RST,ACK SYN -m state --state
NEW -j ACCEPT -A FORWARD -p tcp -m tcp --dport 80 --tcp-flags
FIN,SYN,RST,ACK SYN -m state --state NEW -j ACCEPT -A FORWARD -p
tcp -m tcp --dport 443 --tcp-flags FIN,SYN,RST,ACK SYN -m state
--state NEW -j ACCEPT -A FORWARD -p udp -m udp --dport 53 -m state
--state NEW -j ACCEPT -A FORWARD -p tcp -m tcp --dport 53 -m state
--state NEW -j ACCEPT -A FORWARD -p icmp -m icmp --icmp-type 8 -j
ACCEPT
```



```
-A FORWARD -i ! lo -j LOG --log-prefix "DROP " --log-tcp-options
```

```
--log-ip-options -A OUTPUT -m state --state INVALID -j LOG
--log-prefix "DROP INVALID " --log-tcp-options --log-ip-options -A
OUTPUT -m state --state INVALID -j DROP -A OUTPUT -m state --state
RELATED,ESTABLISHED -j ACCEPT -A OUTPUT -p tcp -m tcp --dport 21
--tcp-flags FIN,SYN,RST,ACK SYN -m state --state NEW -j ACCEPT -A
OUTPUT -p tcp -m tcp --dport 22 --tcp-flags FIN,SYN,RST,ACK SYN -m
state --state NEW -j ACCEPT -A OUTPUT -p tcp -m tcp --dport 25
--tcp-flags FIN,SYN,RST,ACK SYN -m state --state NEW -j ACCEPT -A
OUTPUT -p tcp -m tcp --dport 43 --tcp-flags FIN,SYN,RST,ACK SYN -m
state --state NEW -j ACCEPT -A OUTPUT -p tcp -m tcp --dport 80
--tcp-flags FIN,SYN,RST,ACK SYN -m state --state NEW -j ACCEPT -A
OUTPUT -p tcp -m tcp --dport 443 --tcp-flags FIN,SYN,RST,ACK SYN
-m state --state NEW -j ACCEPT -A OUTPUT -p tcp -m tcp --dport
4321 --tcp-flags FIN,SYN,RST,ACK SYN -m state --state NEW -j
ACCEPT -A OUTPUT -p udp -m udp --dport 53 -m state --state NEW -j
ACCEPT -A OUTPUT -p tcp -m tcp --dport 53 -m state --state NEW -j
ACCEPT -A OUTPUT -p icmp -m icmp --icmp-type 8 -j ACCEPT -A OUTPUT
-o ! lo -j LOG --log-prefix "DROP " --log-tcp-options
--log-ip-options COMMIT # Completed on Sat Apr 14 17:35:22 2007
```

At this point we have a functional iptables policy that maintains a high level of control over the packets that attempt to traverse the firewall inter- faces, and we have a convenient way to rapidly reinstantiate this policy by executing the `iptables-restore` command against the ipt.save file. This has obvious applications for accelerating the system boot cycle, but it is also useful for testing new policies, since it makes it extremely easy to revert to a known-good state. There is one thing missing, however: Altering the iptables policy is most easily accomplished by editing a script instead of by editing the ipt.save file directly (which has a strict syntax requirement that is not as widely known as, say, a Bourne shell script).

Testing the Policy: TCP

Once an iptables policy has been created within the Linux kernel and basic connectivity through the firewall has been verified, it is a good idea to test the policy in order to make sure there are no chinks in the virtual armor. It is most important to test the iptables policy from a host that is external to the local network, because this is the source of the majority of attacks (assuming a huge number of users are not on the internal systems). Effective testing is also important from the internal network, however, since one of the internal hosts could be compromised and then used to attack other internal hosts (including the firewall), even though iptables is protecting the entire network.

Client-side vulnerabilities, such as the Microsoft JPEG vulnerability,[6] make this a

realistic possibility if there are unpatched systems on the internal network.

To begin testing the policy, we first test access to TCP ports that should not be accessible from the either the internal or external networks. Recall that RFC 793 requires a properly implemented TCP stack to generate a reset (RST/ACK[7]) packet if a SYN packet is received on closed port. This provides us with an easy way to verify that iptables is actually blocking packets, since the absence of a RST/ACK packet in response to a connection attempt would indicate that iptables has intercepted the SYN packet within the kernel and has not allowed the TCP stack to generate the RST/ACK back to the client. We randomly select TCP port 5500 to test from both internal and external hosts. The following example illustrates this test and shows that the iptables `INPUT` chain is indeed functioning correctly, since not only are the packets dropped, but the appropriate log messages are also generated. First we test from the ext_scanner system by using Netcat to attempt to connect to TCP port 5500 on the firewall. As expected, the Netcat client just hangs, and on the firewall itself, a log message is generated indicating that iptables inter- cepted and dropped a TCP SYN packet to port 5500:

```
[ext_scanner]$ nc -v 71.157.X.X 5500 [iptablesfw]# tail
/var/log/messages |grep 5500 Apr 14 16:52:43 iptablesfw kernel:
DROP IN=eth0 OUT= MAC=00:13:d3:38:b6:e4:00:30:48:80:4e:37:08:00
SRC=144.202.X.X DST=71.157.X.X LEN=60 TOS=0x00 PREC=0x00 TTL=64
ID=54983 DF PROTO=TCP SPT=59604 DPT=5500 WINDOW=5840 RES=0x00 SYN
URGP=0 OPT (020405B40402080A1E9241460000000001030306)
```

NOTE *The above iptables log message is the first in the book, and you may have trouble mak-*

ing sense of it. I will cover iptables log messages in detail (and with an eye toward recognizing suspicious traffic) in Chapters 2 and 3.

Similarly, we get the same results from the internal network:

```
[int_scanner]$ nc -v 192.168.10.1 5500 [iptablesfw]# tail
/var/log/messages |grep 5500 |tail -n 1 Apr 14 16:55:53
iptablesfw kernel: DROP IN=eth1 OUT=
MAC=00:13:46:3a:41:4b:00:a0:cc:28:42:5a:08:00
SRC=192.168.10.200 DST=192.168.10.1 LEN=60 TOS=0x10
PREC=0x00 TTL=64 ID=4858 DF PROTO=TCP SPT=58715 DPT=5500
WINDOW=5840 RES=0x00 SYN URGP=0 OPT
(020405B40402080A0039F4D30000000001030305)
```

If we had received a RST/ACK packet in either of the tests in the above code example (which would indicate that iptables had not intercepted the SYN packet before it had a chance to interact with the TCP stack running on the firewall), Netcat would have displayed the message `Connection refused`.

[6] See http://www.securityfocus.com/archive/1/375204/2004-09-09/2004-09-15/0 for more information. [7] The details regarding whether or not a RST packet has the ACK bit set are discussed in detail in Chapter 3.

NOTE *It's a good idea to run Nmap against the firewall to rigorously test the iptables policy.*

Nmap offers many different scanning types that assist in making sure that the connection- tracking and filtering capabilities offered by iptables are doing their jobs. For example, sending a surprise FIN packet (see Nmap's `-sF` *scanning mode) against a closed port should not elicit a RST/ACK packet if iptables is working properly. Generating TCP ACK packets that are not part of any established session (Nmap's* `-sA` *mode) should similarly be met with utter silence, because the connection-tracking subsystem is able to discern that such packets are not part of any legitimate TCP session.*

Testing the Policy: UDP

Next, we'll test iptables's ability to filter against UDP ports. Servers that run over UDP sockets exist in a different world than those that run over TCP sockets. UDP is a connectionless protocol, and so there is no notion analo- gous to a TCP handshake or even a scheme to acknowledge data in UDP traffic. Similar constructs such as reliable data delivery can be built in to appli- cations that run over UDP, but this requires application-level modifications, whereas TCP has these features built in for free. UDP simply throws packets out on the network and hopes they reach the intended destination.

To show that iptables is indeed working properly for UDP traffic, we send packets to UDP port 5500 again from both internal and external systems, just as we did for TCP. However, this time, if our UDP packet is not filtered, we should receive an ICMP Port Unreachable message back to our client. This time, we use the hping utility (see http://www.hping.org). In both cases of the external and internal hosts trying to talk to the UDP stack running on the firewall, iptables correctly intercepts the packets. First we test from the exter- nal host:

```
[ext_scanner]# hping -2 -p 5500 71.157.X.X HPING 71.157.X.X (eth0
71.157.X.X): udp mode set, 28 headers + 0 data bytes
[iptablesfw]# tail /var/log/messages |grep 5500 Apr 14 16:58:31
iptablesfw kernel: DROP IN=eth0 OUT=
MAC=00:13:d3:38:b6:e4:00:30:48:80:4e:37:08:00 SRC=144.202.X.X
DST=71.157.X.X LEN=28 TOS=0x00 PREC=0x00 TTL=64 ID=22084
PROTO=UDP SPT=2202 DPT=5500 LEN=8
```

Similarly, we achieve the same result for the internal network:

```
[int_scanner]# hping -2 -p 5500 192.168.10.1 HPING 192.168.10.1
(eth0 192.168.10.1): udp mode set, 28 headers + 0 data bytes
[iptablesfw]# tail /var/log/messages |grep 5500 |tail -n 1 Apr 14
17:00:24 iptablesfw kernel: DROP IN=eth1 OUT=
MAC=00:13:46:3a:41:4b:00:a0:cc:28:42:5a:08:00 SRC=192.168.10.200
DST=192.168.10.1 LEN=28 TOS=0x00 PREC=0x00 TTL=64 ID=35261
```

```
PROTO=UDP SPT=2647 DPT=5500 LEN=8
```


NOTE *This brings up an interesting observation about security: In these tests, any unprivileged user could have used Netcat to listen on TCP or UDP port 5500, but we would have been completely unable to access the server from any IP address that is not explicitly allowed through by the iptables policy. This means that any server started on the system cannot adversely affect the overall security of the system (at least from remote attacks) without also modifying the iptables policy. This is a powerful concept that helps to make the case that a firewall should be deployed on every system; the additional work that is created by having to manage the firewall policy is well worth the effort in the face of risking potential compromise.*

Testing the Policy: ICMP

Finally, we'll test the iptables policy over ICMP. The iptables commands used in the construction of the policy used the `--icmp-type` option to restrict accept- able ICMP packets to just Echo Request packets (the connection-tracking code allows the corresponding Echo Reply packets to be sent so an explicit `ACCEPT` rule does not have to be added to allow such replies). Therefore, iptables should be allowing all Echo Request packets, but other ICMP packets should be met with stark silence. We test this by generating ICMP Echo Reply packets without sending any corresponding Echo Request packets, which should cause iptables to match the packets on the `INVALID` state rule at the beginning of the `INPUT` chain. Again, we turn to hping to test from both the internal and external networks. The first test is to generate an unsolicited ICMP Echo Reply packet from the external network, and we expect that iptables will log and drop the packet in the `INPUT` chain. By examining the iptables log, we see that this is indeed the case (the `DROP INVALID` log prefix is in bold):

```
[ext_scanner]# hping -1 --icmptype echo-reply 71.157.X.X HPING
(eth1 71.157.X.X): icmp mode set, 28 headers + 0 data bytes ---
71.157.X.X hping statistic --- 2 packets transmitted, 0 packets
received, 100% packet loss round-trip min/avg/max = 0.0/0.0/0.0
ms [iptablesfw]# tail /var/log/messages |grep ICMP Apr 14
17:04:58 iptablesfw kernel: DROP INVALID IN=eth0 OUT=
MAC=00:13:d3:38:b6:e4:00:30:48:80:4e:37:08:00 SRC=144.202.X.X
DST=71.157.X.X LEN=28 TOS=0x00 PREC=0x00 TTL=64 ID=44271

PROTO=ICMP TYPE=0 CODE=0 ID=21551 SEQ=0
```

Similarly, the same result is achieved from the internal network:

```
[int_scanner]# hping -1 --icmptype echo-reply 192.168.10.1 HPING
(eth1 192.168.10.1): icmp mode set, 28 headers + 0 data bytes ---
192.168.10.1 hping statistic --- 2 packets transmitted, 0 packets
received, 100% packet loss round-trip min/avg/max = 0.0/0.0/0.0
ms [iptablesfw]# tail /var/log/messages |grep ICMP |tail -n 1 Apr
14 17:06:45 iptablesfw kernel: DROP INVALID IN=eth1 OUT=
MAC=00:13:46:3a:41:4b:00:a0:cc:28:42:5a:08:00 SRC=192.168.10.200
DST=192.168.10.1 LEN=28 TOS=0x00 PREC=0x00 TTL=64 ID=36520
PROTO=ICMP TYPE=0 CODE=0 ID=44313 SEQ=0
```


Concluding Thoughts

This chapter focuses on iptables concepts that are important for the rest of the book and lays a foundation from which to begin discussing intrusion detection and response from an iptables standpoint. We are now armed with a default iptables policy and network diagram that is referenced in several upcoming chapters, and we have seen examples of iptables log messages that illustrate the completeness of the iptables logging format. We are now ready to jump into a treatment of attacks that we can detect—and thwart, as we shall see—with iptables.

2

NETWORK LAYER ATTACKS AND DEFENSE

The network layer—layer three in the OSI Reference Model—is the primary mecha- nism for end-to-end routing and delivery of packet data on the Internet. This book is con- cerned mostly with attacks that are delivered over the IPv4 networking protocol, though many other networking protocols also exist, such as IPX, X.25, and the latent IPv6 protocol.

In this chapter, we'll focus first on how iptables logs network layer packet headers within log message output. Then we will see how these logs can be used to catch suspicious network layer activity.

Logging Network Layer Headers with iptables

With the iptables `LOG` target, firewalls built with iptables have the ability to write

log data to syslog for nearly every field of the IPv4 headers.[1] Because the iptables logging format is quite thorough, iptables logs are well-suited to supporting the detection of many network layer header abuses.

[1] The same is true of IPv6 headers, but IPv6 is not covered in this book.

Logging the IP Header

The IP header is defined by RFC 791, which describes the structure of the header used by IP. Figure 2-1 displays the IP header, and the shaded boxes represent the fields of the header that iptables includes within its log messages. Each shaded box contains the IP header field name followed by the identi- fying string that iptables uses to tag the field in a log message. For example, the Total Length field is prefixed with the string `LEN=` followed by the actual total length value in the packet, and the Time-to-Live (TTL) field is prefixed with `TTL=` followed by the TTL value.

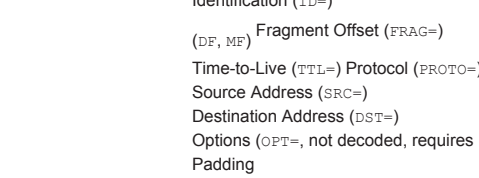
Figure 2-1: The IP header and corresponding iptables log message fields

The dark gray boxes in Figure 2-1 are always logged[2] by iptables. The white boxes denote header fields that are not logged by iptables under any circum- stances. The medium gray box is for the options portion of the IP header. This box is shaded medium gray because iptables only logs IP options if the `--log-ip-options` command-line argument is used when a `LOG` rule is added to the iptables policy.

Here is an example iptables log message generated by sending an ICMP Echo Request from the ext_scanner system toward the iptablesfw system (refer to Figure 1-2):

```
[ext_scanner]$ ping -c 1 71.157.X.X PING 71.157.X.X (71.157.X.X) 56(84) bytes of data. 64
bytes from 71.157.X.X: icmp_seq=1 ttl=64 time=0.171 ms
--- 71.157.X.X ping statistics --- 1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.171/0.171/0.171/0.000 ms [iptablesfw]# tail /var/log/messages | grep
ICMP | tail -n 1 Jul 22 15:01:25 iptablesfw kernel: IN=eth0 OUT=
MAC=00:13:d3:38:b6:e4:00:30:48:80:4e:37:08:00 SRC=144.202.X.X DST=71.157.X.X LEN=84 TOS=0x00
PREC=0x00 TTL=64 ID=0 DF PROTO=ICMP TYPE=8 CODE=0 ID=44366 SEQ=1
```

[2] There is one exception for the IP Fragment Offset—it is only logged by iptables when it is nonzero.



The IP header begins in the log message above with the source IP address (expanded into the standard dotted quad notation).[3] Additional IP header fields such as the destination IP address, TTL value, and the protocol field are in bold. The Type Of Service field (TOS), and the *precedence* and corre- sponding *type* bits are included as separate hexadecimal values to the `TOS` and `PREC` fields. The `Flags` header field in this case is included as the string `DF`, or `Don't Fragment`, which indicates that IP gateways are not permitted to split the packet into smaller chunks. Finally, the `PROTO` field is the protocol encapsulated by the IP header—ICMP in this case. The remaining fields in the log message above include the ICMP `TYPE`, `CODE`, `ID`, and

SEQ values in the ICMP Echo Request packet sent by the ping command, and are not part of the IP header.

Logging IP Options IP options provide various control functions for IP communications, and these functions include timestamps, certain security capabilities, and provisions for special routing features. IP options have a variable length and are used rela- tively infrequently on the Internet. Without IP options, an IP packet header is always exactly 20 bytes long. For iptables to log the options portion of the IP header, use the following command (note the --log-ip-options switch in bold):

```
[iptablesfw]# iptables -A INPUT -j LOG
--log-ip-options
```

The default LOG rules in the policy built by the iptables.sh script in Chap- ter 1 all use the --log-ip-options command-line argument, because IP options can contain information that has security implications.

Now, to illustrate an iptables log message that includes IP options, we once again ping the iptablesfw system, but this time we instruct the ping command to set the timestamp option to tsonly (only timestamp):

```
[ext_scanner]$ ping -c 1 -T tsonly 71.157.X.X
PING 71.157.X.X (71.157.X.X) 56(124) bytes of
data. 64 bytes from 71.157.X.X icmp_seq=1
ttl=64 time=0.211 ms TS: 68579524 absolute
578 0 -578 --- 71.157.X.X ping statistics --- 1 packets
transmitted, 1 received, 0% packet loss, time 0ms rtt
min/avg/max/mdev = 0.211/0.211/0.211/0.000 ms [iptablesfw]# tail
/var/log/messages | grep ICMP Jul 22 15:03:00 iptablesfw kernel:
IN=eth0 OUT= MAC=00:13:d3:38:b6:e4:00:30:48:80:4e:37:08:00
SRC=144.202.X.X DST=71.157.X.X LEN=124 TOS=0x00 PREC=0x00 TTL=64
ID=0 DF OPT (44280D00041670C404167306000000
00000000000000000000000000000000000000000000000000) PROTO=ICMP
TYPE=8 CODE=0 ID=57678 SEQ=1
```

[3] The iptables LOG target automatically converts the integer representation of an IP address within the kernel to the dotted quad notation for readability in the syslog message. There are other instances of such conversions as well, such as for TCP flags, as we will see in Chapter 3. For reference, the kernel portion of the iptables LOG target is implemented within the file linux/net/ ipv4/netfilter/ipt_LOG.c in the kernel sources.

In bold above, the string `OPT` is followed by a long sequence of hexadecimal bytes. These bytes are the

complete IP options included in the IP header, but they are not decoded for us by the iptables `LOG` target; as you'll see in Chapter 7, we'll use psad to make sense of them.

Logging ICMP The iptables `LOG` target has code dedicated to logging ICMP, and since ICMP exists at the network layer,[4] we'll cover it next. ICMP (defined by RFC 792) has a simple header that is only 32 bits wide. Figure 2-2 displays the ICMP header. This header consists of three fields: `type` (8 bits), `code` (8 bits), and a `checksum` (16 bits); the remaining fields are part of the data portion of an ICMP packet.

The specific fields within the data portion depend on the ICMP type and code values. For example, fields associated with an ICMP Echo Request (type 8, code 0) include an ID and a sequence value.

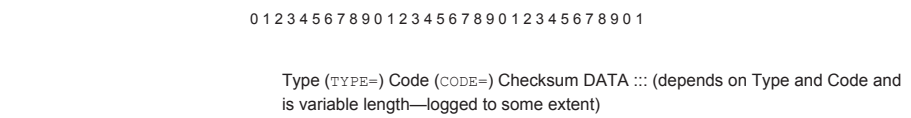


Figure 2-2: The ICMP header and corresponding iptables log message fields

Like the IP header, the `LOG` target always logs the `ICMP` type and `code` fields, and never logs the ICMP `checksum` field. There are no command-line arguments in iptables to influence how the `LOG` target represents fields within the data portion of ICMP packets.[5] The ICMP fields in the first Echo Request packet in this chapter appear starting in the last line below:

```
Jul 22 15:01:25 iptablesfw kernel: IN=eth0 OUT=
MAC=00:13:d3:38:b6:e4:00:30:48:80:4e:37:08:00 SRC=144.202.X.X
DST=71.157.X.X LEN=84 TOS=0x00 PREC=0x00 TTL=64 ID=0 DF
PROTO=ICMP TYPE=8 CODE=0 ID=44366 SEQ=1
```

Network Layer Attack Definitions

We define a network layer attack as a packet or series of packets that abuses the fields of the network layer header in order to exploit a vulnerability in the network stack implementation of an end host, consume network layer resources, or conceal the delivery of exploits against higher layers.

[4] Contrary to the tendency some have of lumping ICMP into the bucket reserved for transport layer protocols such as TCP and UDP, ICMP is considered a network layer protocol. See W. Richard Stevens' book *TCP/IP Illustrated, Volume 1*, page 69 (Addison-Wesley, 1994). [5] An examination of the `switch` statement, beginning at line 249 of the `LOG` target source code in the Linux kernel (see the file

linux/net/ipv4/netfilter/ipt_LOG.c), sheds light on this.

Network attacks fall into one of three categories:

Header abuses Packets that contain maliciously constructed, broken, or falsified network layer headers. Examples include IP packets with spoofed source addresses and packets that contain unrealistic fragment offset values.
Network stack exploits Packets that contain specially constructed components designed to exploit a vulnerability in the network stack implementation of an end host. That is, the code dedicated to the processing of network layer information is itself the target. A good example is the Internet Group Management Protocol (IGMP) Denial of Service (DoS) vulnerability discovered in the Linux kernel (versions 2.6.9 and earlier).[6] **Bandwidth saturation** Packets that are designed to saturate all available bandwidth on a targeted network. A Distributed Denial of Service (DDoS) attack sent over ICMP is a good example.

NOTE *Although this chapter focuses on techniques for abusing the network layer, it is important to note that many of these techniques can be combined with attacks at other layers. For example, an application layer attack (say, one that exploits a buffer overflow vulnerability) can be sent over fragmented IP packets in an effort to evade intrusion detection systems. In this case, the real attack exploits an application layer vulnerability but is delivered using a network layer technique called* fragmentation *that makes the application layer attack more difficult to detect.*

Abusing the Network Layer

The network layer's ability to route packets to destinations around the world provides the ability to attack targets worldwide as well. Because IPv4 does not have any notion of authentication (this job is left to the IPSec protocol or to mechanisms at higher layers), it is easy for an attacker to craft IP packets with manipulated headers or data and splat them out onto the network. While such packets may be filtered by an inline filtering device such as a firewall or router with an Access Control List (ACL) before ever reaching their intended target, they frequently are not.

Nmap ICMP Ping

When Nmap is used to scan systems that are not on the same subnet, host discovery is performed by sending an ICMP Echo Request and a TCP ACK to port 80 on the targeted hosts. (Host discovery can be disabled with the Nmap `-P0` command-line argument, but it is enabled by default.) ICMP Echo Requests generated by Nmap differ from the Echo Requests generated by the ping pro- gram in that Nmap Echo

Requests do not include any data beyond the ICMP

[6] The Linux kernel IGMP vulnerability is assigned the designation CAN-2004-1137 in the Common Vulnerabilities and Exposures (CVE) database, which is one of the best tracking mechanisms for vulnerabilities available today. See http://cve.mitre.org/cve for more information.

header. Therefore, if such a packet is logged by iptables, the IP length field should

be 28 (20 bytes for the IP header without options, plus 8 bytes for the ICMP header, plus 0 bytes for data, as shown in bold):

```
[ext_scanner]# nmap -sP 71.157.X.X [iptablesfw]# tail
/var/log/messages | grep ICMP Jul 24 22:29:59 iptablesfw kernel:
IN=eth0 OUT= MAC=00:13:d3:38:b6:e4:00:30:48:80:4e:37:08:00
SRC=144.202.X.X DST=71.157.X.X LEN=28 TOS=0x00 PREC=0x00 TTL=48
ID=1739 PROTO=ICMP TYPE=8 CODE=0 ID=15854 SEQ=62292
```

NOTE *The ping program can also generate packets without application layer data by using the `-s 0` command-line argument to set a zero size on the payload, but by default the ping program includes a few tens of bytes of payload data.*

While not including application layer data in an ICMP packet is not in and of itself an abuse of the network layer, if you see such packets in conjunction with packets that indicate activities such as port scans or port sweeps (see Chap- ter 3), it is a good bet that someone is performing reconnaissance against your network with Nmap.

IP Spoofing

Few terms in computer security give rise to more confusion and hyperbole than *spoofing*, specifically *IP spoofing*. A *spoof* is a hoax or prank, and IP spoofing means to deliberately construct an IP packet with a falsified source address.

NOTE *We carve out an exception here for Network Address Translation (NAT) operations on IP packets which alter source addresses (such as commonly provided by firewalls to shield internal networks behind a single external address). Not to be confused with IP spoofing, NAT is a legitimate networking function, whereas concealing an attack with a falsified source address is not.*

When it comes to communications over IP, there is no built-in restriction on the source address of a packet. By using a raw socket (a low-level program- ming API to craft packets according to certain criteria), an IP packet can be sent with an arbitrary source address. If the source address is nonsensical in the context of the local network (for example, if the source is an IP on Verizon's network but the packet is really being sent from Comcast's network), the packet is said to be *spoofed*. Administrators can take steps to configure routers and firewalls to not forward packets with source addresses outside of internal network ranges (so spoofed packets would never make it out), but many networks have no such controls. The default iptables policy discussed in Chapter 1 has anti-spoofing rules built in.

From a security perspective, the most important thing to know about spoofed packets (and IP packets in general) is that it is impossible to trust the source address. In fact, sometimes a complete attack can be delivered in a single spoofed packet (see the Witty worm discussion in Chapter 8).

NOTE *Any packet with a spoofed source address is purely "fire and forget," since any response to the packet from the target is directed back to the fake, spoofed address. Some solace can be had, though, from recognizing that any protocol that requires bidirectional traffic, such as TCP at the transport layer, will not function over spoofed IP addresses.*[7]

Many pieces of security software (both offensive and defensive) include the ability to spoof source IP addresses. Distributed Denial of Service (DDoS) tools generally regard IP spoofing as a necessity, and well-known tools such as hping and Nmap can spoof source addresses as well.

IP SPOOFING WITH PERL

Crafting a packet with a spoofed source address is trivially easy using a tool such as hping, or with your own spoofing tool. Below is a simple Perl snippet that builds a UDP datagram with a spoofed source address and includes application layer data of your choosing (the "abuse" part of this example is the spoofed source address). The script uses the `Net::RawIP` Perl module; the source IP address is read from the command line at , and then it is set within the IP header at :

```
#!/usr/bin/per
l -w

use Net::RawIP; use strict;
my $src = $ARGV[0] or
&usage(); my $dst =
$ARGV[1] or &usage(); my
$str = $ARGV[2] or
&usage();
    my $rawpkt = new
  Net::RawIP({ ip => {
 saddr => $src, daddr =>
    $dst }, udp =>{}}
 );$rawpkt->set({ ip =>
 { saddr => $src, daddr
   => $dst }, udp => {
source => 10001, dest => 53, data => $str, } }); $rawpkt->send();
    print '[+] Sent ' . length($str) . " bytes of
    data...\n"; exit 0; sub usage() { die "usage:
    $0 <src> <dst> <str>"; }
```

IP

Fragmentation

The ability to split IP packets into a series of smaller packets is an essential feature of IP. The process of splitting IP packets, known as *fragmentation*, is necessary whenever an IP packet is routed to a network where the data link

[7] Successful TCP sequence prediction attacks can allow TCP connections to be torn down or data to be injected into existing connections from spoofed sources.

MTU size is too small to accommodate the packet. It is the responsibility of any

router that connects two data link layers with different MTU sizes to ensure that IP packets transmitted from one data link layer to another never exceed the MTU. The IP stack of the destination host reassembles the IP fragments in order to create the original packet, at which point an encapsulated protocol within the packets is handed up the stack to the next layer. IP fragmentation can be used by an attacker as an IDS evasion mechanism by constructing an attack and deliberately splitting it over multiple IP fragments. Any fully implemented IP stack can reassemble fragmented traffic, but in order to detect the attack, an IDS also has to reassemble the traffic with the same algorithm used by the targeted IP stack. Because IP stacks implement reassembly algorithms slightly differently (e.g., for duplicate fragments, Cisco IOS IP stacks reassemble traffic according to a last fragment policy, whereas Windows XP stacks reassemble according to a first fragment policy), this creates a challenge for an IDS.[8] The gold standard for generating fragmented traffic is Dug Song's fragroute tool (see http://www.monkey.org).

Low TTL Values

Any IP router is supposed to decrement the TTL value in the IP header by one[9] every time an IP packet is forwarded to another system. If packets appear within your local subnet with a TTL value of one, then someone is most likely using the traceroute program (or a variant such as tcptraceroute) against an IP address that either exists in the local subnet or is in a subnet that is routed through the local subnet. Usually this is simply someone troubleshooting a network connectivity problem, but it can also be an instance of someone performing reconnaissance against your network in order to map out hops to a potential target.

NOTE *Packets destined for multicast addresses (all addresses within the range 224.0.0.0 through 239.255.255.255, as defined by RFC 1112) commonly have TTL values set to one. So if the destination address is a multicast address, it is likely that such traffic is not associated with network mapping efforts with traceroute and is just legitimate multicast traffic.*

A UDP packet produced by traceroute is logged as follows by iptables (note the TTL in bold):

```
Jul 24 01:10:55 iptablesfw kernel: DROP IN=eth0 OUT=
MAC=00:13:d3:38:b6:e4:00:13:46:c2:60:44:08:00 SRC=144.202.X.X
DST=71.157.X.X LEN=40 TOS=0x00 PREC=0x00 TTL=1 ID=44081 PROTO=UDP
SPT=54522 DPT=33438 LEN=20
```

[8] Taking a host-centric view of intrusion detection is known as *target-based intrusion detection*, which allows an IDS to factor in implementation details of target systems; more on this in Chapter 8. [9] It is

possible for a router to decrement the TTL value by two or more if the number of seconds the router holds onto the packet before forwarding it is greater than one second. RFC 791 states that a router must decrement the TTL by *at least* one.

CONCEALING AN ATTACK WITH FRAGMENTS AND TARGETED TTLS

Routing path information is useful for concealing network attacks with fragment reassembly tricks. For example, suppose that an attacker sees that a router exists in front of a host (as determined with traceroute), and that the attacker also suspects that an IDS is watching the subnet that is in front of the host subnet. If this is the case, the host can be targeted with an attack that is fragmented over three IP packets (let's call them f1, f2, and f3), but in such a way that the attack is not detected by the IDS. The attacker can accomplish this by creating a duplicate of the second fragment (f2), replacing its payload with dummy data, and reducing its TTL to an initial value that is just large enough to get the packet to the router with a TTL of one. Let's call this packet f2'. Next, the attacker sends the first fragment (f1), followed by this new fragment (f2'), followed by f3, and finally, the original f2 fragment. Thus, the IDS (which is in front of the router) sees all four fragments, but f3 completes the set of fragments and hence the IDS reassembles them as f1 + f2' + f3.

Recall that f2' contains dummy data, so these three fragments together do not look like an attack to the IDS. Meanwhile, f2' hits the router and gets dropped because its TTL value is decremented to zero before it is forwarded, so the target IP address never sees f2'. However, the host has seen fragments f1 and f3, but it can't reassemble them to anything meaningful without the original f2, so it waits for it.

When f2 finally arrives (remember that the attacker sent it last), the target host is hit with the real attack after the host finally reassembles all three fragments. This technique was first proposed in "Bro: A System for Detecting Network Intruders in Real-Time" by Vern Paxson (see http://www.icir.org/vern/papers/bro-CN99.html); it provides a clever way to utilize the network layer to hide attacks from network intrusion detection systems.

NOTE *Another suspicious TTL value for any packet on the local subnet is a TTL of zero. Such a packet can only exist if there is either a severely buggy router that forwarded the packet into the subnet or the packet originated from a system on the same subnet.*

The Smurf Attack

The Smurf attack is an old but elegant technique whereby an attacker spoofs ICMP Echo Requests to a network broadcast address. The spoofed address is the intended target, and the goal is to flood the target with as many ICMP Echo Response packets as possible from systems that respond to the Echo Requests over the broadcast address. If the network is functioning without controls in place against these ICMP Echo Requests to broadcast addresses (such as with the `no ip directed-broadcast` command on Cisco routers), then all hosts that receive the

Echo Requests will respond to the spoofed source address. By using the broadcast address of a large network, the attacker hopes to magnify the number of packets that are generated against the target.

The Smurf attack is outdated when compared to tools that perform DDoS attacks (discussed below) with dedicated control channels and for which there is no easy router configuration countermeasure. Still, it is worth mentioning, because the Smurf attack is so easy to perform and the original source code is readily available (see http://www.phreak.org/archives/exploits/ denial/smurf.c).

DDoS

Attacks

A DDoS attack at the network layer utilizes many systems (potentially thousands) to simultaneously flood packets at target IP addresses. The goal of such an attack is to chew up as much bandwidth on the target network as possible with garbage data in order to edge out legitimate communications. DDoS attacks are among the more difficult network layer attacks to combat because so many systems are connected via broadband to the Internet. If an attacker succeeds at compromising several systems with fast Internet connec- tions, it is possible to mount a damaging DDoS attack against most sites.

Because the individual packets created by a DDoS agent can be spoofed, it is generally futile to assign any value to the source IP address of such packets by the time the packet reaches the victim.

For example, according to the Snort signature ruleset (discussed in later chapters), the Stacheldraht DDoS agent (see http://staff.washington.edu/ dittrich) spoofs ICMP packets from the IP address 3.3.3.3. If you see packets with the source IP address set to 3.3.3.3 and the destination IP address set to an external address, you know that a system on your local network has become a Stacheldraht zombie. A packet sent from Stacheldraht would look similar to the following when logged by iptables. (The source IP address 3.3.3.3 at , the ICMP type of zero at , and the ICMP ID of 666 at come from Snort rule ID 224):

```
Jul 24 01:44:04 iptablesfw kernel: SPOOFED PKT IN=eth0 OUT=
MAC=00:13:d3:38:b6:e4:00:13:46:c2:60:44:08:00 SRC=3.3.3.3
DST=71.157.X.X LEN=84 TOS=0x00 PREC=0x00 TTL=63 ID=0 DF
PROTO=ICMP
 TYPE=0 CODE=0 ID=666
 SEQ=1
```

In general, it is more effective to try to detect the control communications associated with DDoS agents than to detect the flood packets themselves. For example, detecting commands sent from control nodes to zombie nodes over obscure port numbers is a good strategy (several signatures in the Snort rule- set look for communications of this type—see the dos.rules file in the Snort signature set). This can also yield results when removing DDoS agents from a network, because control communications can help point the way to infected systems.

Linux Kernel IGMP Attack

A good example of an attack against the code responsible for processing network layer communications is an exploit for a specific vulnerability in the Internet Group Management Protocol (IGMP) handling code in the Linux kernel. Kernel versions from 2.4.22–2.4.28, and 2.6–2.6.9 are vulnerable and can be exploited both remotely and by local users (some security vulnerabil- ities are only locally exploitable, so this is a nasty bug). A successful exploit over the network from a remote system could result in a kernel crash, as discussed in more detail at http://isec.pl/vulnerabilities/isec-0018-igmp.txt. Kernel code sometimes contains

security bugs, and these bugs can exist all the way down at the network layer processing code or within device drivers.

Network Layer Responses

Agreeing on definitions for network layer responses is as useful as agreeing on definitions for network layer attacks. Because such responses should not involve information that resides at the transport layer or above, we are limited to the manipulation of network layer headers in one of three ways:

A filtering operation conducted by a device such as a firewall or router to block the source IP address of an attacker Reconfiguration of a routing protocol to deny the ability of an attacker to route packets to an intended target by means of *route blackholing*— packets are sent into the void and are never heard from again Applying thresholding logic to the amount of traffic that is allowed to pass through a firewall or router based on utilized bandwidth

A *response* that is purely at the network layer can be used to combat an attack that is *detected* at the application layer, but such a response should not involve things like generating a TCP RST packet for example—this would be a transport layer response, as we'll see in Chapter 3.

Network Layer Filtering Response

After an attack is detected from a particular IP address, you can use the following iptables rules as a network layer response that falls into the filtering category. These rules are added to the `INPUT`, `OUTPUT`, and `FORWARD` chains; they block all communications (regardless of protocol or ports) to or from the IP address 144.202.*X*.*X*:

```
[iptablesfw]# iptables -I INPUT 1 -s 144.202.X.X
-j DROP [iptablesfw]# iptables -I OUTPUT 1 -d
144.202.X.X -j DROP [iptablesfw]# iptables -I
FORWARD 1 -s 144.202.X.X -j DROP [iptablesfw]#
iptables -I FORWARD 1 -d 144.202.X.X -j DROP
```

There are two rules in the `FORWARD` chain to block packets that originate from 144.202.*X*.*X* (`-s 144.202.X.X`) as well as responses from internal systems that are destined for 144.202.*X*.*X* (`-d 144.202.X.X`). If you use iptables as your network sentry, then the above rules provide an effective network choke point against the 144.202.*X*.*X* address.

Network Layer Thresholding Response

Applying thresholding logic to iptables targets is accomplished with the iptables `limit` extension. For example, the `limit` extension can be used within an `ACCEPT` rule to limit the number of packets accepted from a specific source address within a given window of time. The following iptables rules restrict the policy to only accept 10 packets per second to or from the 144.202.*X.X* IP address.


```
[iptablesfw]# iptables -I INPUT 1 -m limit --limit 10/sec -s
144.202.X.X -j ACCEPT [iptablesfw]# iptables -I INPUT 2 -s 144.202.X.X
-j DROP [iptablesfw]# iptables -I OUTPUT 1 -m limit --limit 10/sec -d
144.202.X.X -j ACCEPT [iptablesfw]# iptables -I OUTPUT 2 -d 144.202.X.X
-j DROP [iptablesfw]# iptables -I FORWARD 1 -m limit --limit 10/sec -s
144.202.X.X -j ACCEPT [iptablesfw]# iptables -I FORWARD 2 -s 144.202.X.X
-j DROP [iptablesfw]# iptables -I FORWARD 1 -m limit --limit 10/sec -d
144.202.X.X -j ACCEPT [iptablesfw]# iptables -I FORWARD 2 -d 144.202.X.X
-j DROP
```

For each `ACCEPT` rule above that uses the `limit` match, there is also a corresponding `DROP` rule. This accounts for packets levels that exceed the 10-per-second maximum permitted by the `limit` match; once the packet levels are higher than this threshold, they no longer match on the `ACCEPT` rule and are then compared against the remaining rules in the iptables policy. It is frequently better to just refuse to communicate with an attacker alto- gether than to allow even thresholded rates of packets through.

You can also use the `limit` match to place thresholds on the number of iptables log messages that are generated by default logging rules. However, unless disk space is a concern, applying a limit threshold to a `LOG` rule is not usually necessary, because the kernel uses a ring buffer internally within the `LOG` target so that log messages are overwritten whenever packets hit a `LOG` rule faster than they can be written out via syslog.

Combining Responses Across Layers

Responses can be combined across layers, just as attacks can be. For example, a firewall rule could be instantiated against an attacker at the same time that a TCP RST is sent using a combination of tools like fwsnort and psad (see Chapter 11).

One way to knock down a malicious TCP connection would be to use the iptables `REJECT` target and then instantiate a persistent blocking rule against the source address of the attack. The persistent blocking rule is the network layer response, which prevents any further communication from the attacker's current IP address with the target of the initial attack.

Although this may sound effective, note that a blocking rule in a firewall can frequently be circumvented

by an attacker routing attacks over the The Onion Router (Tor) network.[10] By sending an attack over Tor, the source address of the attack is not predictable by the target.

[10] Tor anonymizes network communications by sending packets through a cloud of nodes called *onion routers* in an encrypted and randomized fashion. Tor only supports TCP, so it cannot be used to anonymize attacks over other protocols such as UDP.

The same is true for attacks where the source IP address is spoofed by the attacker. Spoofed attacks do not require bidirectional communication, and so it is risky to respond to them; doing so essentially gives control to the attacker over who gets blocked in your firewall! It is unlikely that all important IP addresses (such as DNS servers, upstream routers, remote VPN tunnel terminations, and so on) are whitelisted in your firewall policy, and so giving this control to an attacker is risky. Some of the suspicious traffic examples earlier in this chapter, such as spoofed UDP strings, packets with low TTL values, and Nmap ICMP Echo Requests, are perfect examples of traffic that it is *not* a good idea to actively respond to.

As we will see in later chapters, there are only a few classes of traffic that are best met with automated responses.

3

TRANSPORT LAYER ATTACKS AND DEFENSE

The transport layer—layer four in the OSI Reference Model—provides data delivery, flow control, and error recovery services to end hosts on the Internet. The two primary transport layer protocols we are concerned with are the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP).

TCP is a connection-oriented protocol. This means that the client and server negotiate a set of parameters that define how data is transferred before any data is exchanged, and that there is a clear demarcation of the start and end of a connection. TCP transfers data between two nodes in a reliable, in-order fashion, which frees application layer protocols from having to build in this functionality themselves.[1]

In contrast, UDP is a connectionless protocol. As a connectionless pro- tocol, there is no guarantee that data ever reaches its intended destination,

[1] Technically, the transport layer interacts with the session layer above and network layer below in the OSI Reference Model, but it is usually more useful to think of the session layer as subsumed within the application layer (along with the presentation layer).

and there is also no guarantee about the shape of the data that does make it through (even the calculation of the checksum in the UDP header is optional unlike in TCP). Applications that transmit data over UDP sockets can choose to implement additional mechanisms to transmit data reliably, but such func- tionality must be built in to the application layer when UDP sockets are used.

We'll focus first in this chapter on how iptables represents transport layer information within log message output. We'll then see how these logs can catch suspicious transport layer activity.

Logging Transport Layer Headers with iptables

The iptables `LOG` target has extensive machinery for logging TCP and UDP headers. The TCP header is far more complex than the UDP header, and some TCP header fields are logged only if specific command-line arguments are supplied to iptables when a `LOG` rule is added to the iptables policy.

Logging the TCP Header

The TCP header is defined in RFC 793, and the length of the header for any particular TCP segment[2] varies depending on the number of options that are included. The length of the header, excluding the options (which is the only variable-length field), is always 20 bytes. In an iptables log message, each field in the TCP header is prefixed with an identifying string, as shown in Figure 3-1.

Figure 3-1: The TCP header and iptables log message fields

All dark gray boxes in Figure 3-1 are always included within an iptables log message of a TCP packet; the fields shaded in lighter gray are included only if the specified command-line argument is given to iptables. The white boxes are never logged by iptables.

The `LOG` rule in the `INPUT`, `OUTPUT`, and `FORWARD` chains included in the default iptables policy in Chapter 1 are all built with the `--log-tcp-options` argument, so each log message contains a blob of hexadecimal codes when- ever a TCP segment contains options. This chapter assumes that the default iptables policy implemented by the iptables.sh script from Chapter 1 is run- ning on the iptablesfw system depicted in Figure 3-2. (This diagram is identical to Figure 1-2 and is duplicated here for convenience.)

[2] Although the technical term for a unit of TCP information is a TCP *segment*, many people informally refer to TCP *packets* instead (*packets* is technically a term reserved for the network layer), and I use this colloquialism also. The same logic applies to UDP *datagrams*—it is more convenient to refer to UDP *packets*.

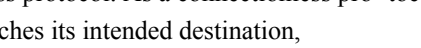


Figure 3-2: Default network diagram

To illustrate TCP options included within an iptables log message, we attempt to initiate a TCP connection to port 15104 from the ext_scanner system to the iptablesfw system.

Because the default policy does not allow communications with port 15104, the initial SYN packet is intercepted by the default iptables `LOG` and `DROP` rules. The tags iptables associates with each field of the TCP header are shown in bold below, starting with the source port (`SPT`) and ending with the options portion of the header (`OPT`):

```
[ext_scanner]$ nc -v 71.157.X.X 15104 [iptablesfw]# tail /var/log/messages | grep 15104 Jul 12
15:10:22 iptablesfw kernel: DROP IN=eth0 OUT= MAC=00:13:d3:38:b6:e4:00:30:48:80:4e:37:08:00
SRC=144.202.X.X DST=71.157.X.X LEN=60 TOS=0x00 PREC=0x00 TTL=64 ID=18723 DF PROTO=TCP
SPT=47454 DPT=15104 WINDOW=5840 RES=0x00 SYN URGP=0 OPT (020405B40402080A30820
48C0000000001030306)
```

To have iptables include TCP sequence and acknowledgment values, use the `--log-tcp-sequence` argument (see the sections in bold below):

```
[iptablesfw]# iptables -I INPUT 1 -p tcp --dport 15104 -j LOG --log-tcp-options
--log-tcp-sequence [ext_scanner]$ nc -v 71.157.X.X 15104 [iptablesfw]# tail /var/log/messages
| grep 15104 Jul 12 15:33:53 iptablesfw kernel: IN=eth0 OUT=
MAC=00:13:d3:38:b6:e4:00:30:48:80:4e:37:08:00 SRC=144.202.X.X DST=71.157.X.X LEN=60 TOS=0x00
PREC=0x00 TTL=64 ID=62378 DF PROTO=TCP SPT=54133 DPT=15104 SEQ=3180893451 ACK=0 WINDOW=5840
RES=0x00 SYN URGP=0 OPT (020405B40402080A308766A10000000001030306)
```

LAN
LAN Desktop 192.168.10.0/24
Hostname: lan_client
Internet

iptables Firewall Hostname: iptablesfw

192.168.10.50

71.157.X.X (eth0) 192.168.10.1 (eth1)
Webserver Hostname: webserver

192.168.10.3 DNS Server

Hostname: dnsserver 192.168.10.4
Internal Scanner Hostname: int_scanner 192.168.10.200

Logging the UDP Header

The UDP header is defined in RFC 768. It is only eight bytes long and has no variable length fields (see Figure 3-3). Since there are no special command-line arguments to influence how a UDP header is represented by the `LOG` target, iptables always logs UDP headers in the same way.

0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1

Figure 3-3: The UDP header and iptables log message fields

Even though the default `LOG` rules in the iptables policy discussed in Chapter 1 use the `--log-tcp-options` argument, if a UDP packet hits one of these rules, iptables does the right thing and only logs information that is actually in the packet; it won't attempt to log the options portion of a TCP header that does not exist. The UDP checksum is never logged, but the remaining three fields (`SPT`, `DPT`, and `LEN`) are all included:

```
[ext_scanner]$ echo -n "aaaa" | nc -u 71.157.X.X 5001 [iptablesfw]# tail /var/log/messages |
grep 5001 Jul 12 16:27:08 iptablesfw kernel: DROP IN=eth0 OUT=
MAC=00:13:d3:38:b6:e4:00:30:48:80:4e:37:08:00 SRC=144.202.X.X DST=71.157.X.X LEN=33 TOS=0x00
PREC=0x00 TTL=64 ID=38817 DF PROTO=UDP SPT=44595 DPT=5001 LEN=12
```

NOTE *The UDP `LEN` field in the iptables log message above includes the length of the UDP header plus the length of the application layer data. In this case, the application layer data consists of the four bytes `"aaaa"`, so adding this to the length of the UDP header (eight bytes) yields a total of 12 bytes. The `-n` command-line argument to the `echo` command instructs it not to add a trailing newline character. Had this argument not been used, the value of the `LEN` field would have been 13 to accommodate the additional byte.*

Transport Layer Attack Definitions

Like the definition of a network layer attack (given in Chapter 2), we define a *transport layer attack* as a packet or series of packets that abuses the fields of the transport layer header in order to exploit either a vulnerability or error condition in the transport stack implementation of an end host.

Transport layer attacks fall into one of the following three categories: **Connection resource exhaustion** Packets that are designed to saturate all available resources for servicing new connections on a targeted host or set of hosts. A good example is a DDoS attack in the form of a SYN flood.

Source Port (SPT=)
Destination Port (DPT=) Length (LEN=) Checksum

Header abuses Packets that contain maliciously constructed, broken, or falsified transport layer headers. A good example is a forged RST packet designed to tear down a TCP connection. We lump port scans (discussed below) into this category as well, although a scan *by itself* is not malicious.
Transport stack exploits Packets that contain transport layer stack exploits for vulnerabilities in the stack of an end host. That is, the kernel code dedicated to the processing of transport layer information is itself the target. A good example (especially in the context of this book) is an exploit announced in 2004 for a vulnerability in the Netfilter TCP options processing code (this bug was quickly fixed by the Netfilter project, so any recent version of the kernel is not vulnerable). While this does not exploit the TCP stack itself, it exploits code that is directly hooked into the stack via the Netfilter framework.

Abusing the Transport Layer

Because the transport layer is, in a sense, the last gateway before communicat- ing up the stack with a networked application, it's a juicy target for an attacker. Much of the suspicious activity that involves transport layer information falls into the category of reconnaissance efforts instead of outright attacks.

Port Scans

A port scan is a technique used to interrogate a host in order to see what TCP or UDP services are accessible from a particular IP address. Scanning a sys- tem can be an important step along the way toward a successful compromise, because it gives information to an attacker about services that may be accessed and attacked. That said, a port scan can also be an important step to just seeing what services are available to talk to; there is nothing inherently malicious about a port scan by itself. You can liken a port scan to a person knocking on all the doors of a house. For any given door, if someone answers and the person just says, "Hello, nice to meet you," and then walks away, no harm is done. While the repeated knocking may be suspicious, a crime has probably not been committed unless the person attempts to enter the house. Still, if someone were to knock on all the doors of *my* house, I would want to know about it, because it may be a sign of someone collecting information about the best way to break in. Similarly, it's a good idea to detect port scans (subject to a tuning exercise to reduce false positives), and most network intrusion detec- tion systems offer the ability to send alerts when systems are hit with a scan.

Matching Port Scans to Vulnerable Services A port scan does not have to involve an exhaustive test for every possible port on a target system.[3] If an attacker is skilled at compromising, say, OpenSSH 3.3 and BIND 4.9 servers, then it is of little use to find out if the remaining

[3] The source and destination port fields in the TCP and UDP headers are 16 bits wide, so there are 65,536 (2^16) total ports (including port 0, which can be scanned by Nmap).

65,533[4] ports also have servers bound to them. Furthermore, generating a noisy

scan to test all ports on a system is a good way to set off IDS alarm bells, because it is much more likely that any reasonable port scan thresholds would be tripped. As an attacker, it is better to not call unnecessary attention to oneself. To make it even more difficult for an IDS to determine the real source of a scan, an attacker can also use Nmap's decoy (`-D`) option. This allows a port scan to be duplicated from several spoofed source addresses, so it appears to the target system as though it is being scanned by several independent sources simultaneously. The goal is to make it harder for any security administrator who may be watching IDS alerts to work out the real source of a scan.

TCP Port Scan Techniques Port scans of TCP ports can be accomplished using a surprising number of techniques. Each of these techniques looks slightly different on the wire as packets traverse a network, and we dedicate the next few sections (beginning with "TCP connect() Scans" and ending with "TCP Idle Scans" on page 59) to illustrating the major scanning techniques. Fortunately, the unequaled Nmap scanner (see http://www.insecure.org) has automated each of these techniques for us, and we use Nmap for all scan examples in this chapter. We launch scans against the iptablesfw system with the default iptables policy active (see Figure 3-2), and we will discuss the Nmap port-scanning techniques listed below:

TCP `connect()` scan—(Nmap `-sT`) TCP SYN or half-open scan—(Nmap `-sS`) TCP FIN, XMAS, and NULL scans—(Nmap `-sF`, `-sX`, `-sN`) TCP ACK scan—(Nmap `-sA`) TCP idle scan—(Nmap `-sI`) UDP scan—(Nmap `-sU`)

In each of the following scans, the Nmap `-P0` command line option is used to force Nmap to skip determining whether the `iptablesfw` system is up (i.e., host discovery is omitted) before sending a scan. From Nmap's perspec- tive, each scanned port can be in one of three states:

open There is a server bound to the port, and it is accessible. **closed** There is no server bound to the port. **filtered** There may be a server bound to the port, but attempts to communicate with it are blocked, and Nmap cannot determine if the port is open or closed.

TCP connect() Scans When a normal client application attempts to communicate over a network to a server that is bound to a TCP port, the local TCP stack interacts with the

[4] Even though port zero can be scanned by Nmap, operating systems do not allow servers to `bind()` to port zero.

remote stack on behalf of the client. Before any application layer data is transmitted, the two stacks must negotiate the parameters that govern the conversation that is about to take place between the client and server. This

negotiation is the standard TCP three-way handshake and requires three packets, as shown in Figure 3-4.

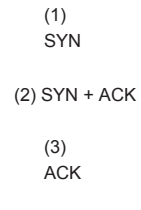


Figure 3-4: TCP three-way handshake

The first packet, SYN (short for *synchronize*), is sent by the client to the server. This packet advertises the desired initial sequence number (among other things, such as the TCP window size and options such as whether Selective Acknowledgment is permissible) used for tracking data trans- mission across the TCP session to the server. If the SYN packet reaches an open port, the server TCP stack responds with a SYN/ACK to acknowledge the receipt of the initial sequence value from the client and to declare its own sequence number back to the client. The client receives the SYN/ACK and responds with an acknowledgment to the server. At this point, both sides have agreed on the connection parameters (including the initial sequence numbers), and the connection state is defined as established and ready to transfer data.

In the context of the TCP `connect()` scan, the scanner sends both the SYN and the ending ACK packet for each scanned port. Any normal user can scan a remote system in this mode with Nmap; no special privileges are required.

Below are some of the iptables log messages displayed from a SYN scan along with the Nmap output. You can see that the http and https ports are open, and the options portion of the SYN packet contains a substantial number of options:

```
[ext_scanner]$ nmap -P0 -sT 71.157.X.X Starting Nmap 4.01 (
http://www.insecure.org/nmap/ ) at 2007-07-03 00:32 EDT
Interesting ports on 71.157.X.X: (The 1670 ports scanned but not
shown below are in state: filtered) PORT STATE SERVICE 80/tcp
open http 443/tcp open https Nmap finished: 1 IP address (1 host
up) scanned in 30.835 seconds

[iptablesfw]# grep SYN /var/log/messages | tail -n 1 Jul 3
00:32:32 iptablesfw kernel: DROP IN=eth0 OUT=
MAC=00:13:d3:38:b6:e4:00:30:48:80:4e:37:08:00 SRC=144.202.X.X
DST=71.157.X.X LEN=60 TOS=0x00 PREC=0x00 TTL=64 ID=65148 DF
PROTO=TCP SPT=43237 DPT=653 WINDOW=5840 RES=0x00 SYN URGP=0 OPT
(020405B40402080A362957720000000001030306)
```

TCP SYN or Half-Open Scans A SYN or half-open scan is similar to a

`connect()` scan in that the scanner sends a SYN packet to each TCP port in an effort to elicit a SYN/ACK or RST/ACK response that will show if the targeted port is open or closed. However, the scanning system never completes the three-way handshake because it deliber- ately fails to return the ACK packet to any open port that responds with a SYN/ACK. Therefore, a SYN scan is also known as a half-open scan because three-way handshakes are never given a chance to gracefully complete, as depicted in Figure 3-5.

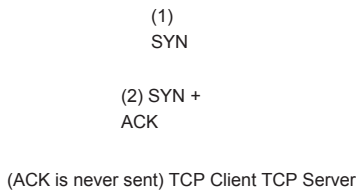


Figure 3-5: TCP half-open scan

A SYN scan cannot be accomplished with the `connect()` system call because that call invokes the vanilla TCP stack code, which will respond with an ACK for each SYN/ACK received from the target. Hence, every SYN packet sent in a SYN scan must be crafted by a mechanism that bypasses the TCP stack altogether. This is commonly accomplished by using a raw socket to build a data structure that mimics a SYN packet when placed on the wire by the OS kernel.

RAW SOCKETS AND UNSOLICITED SYN/ACKS

Using a raw socket to craft a TCP SYN packet toward a remote system instead of using the `connect()` system call brings up an interesting issue. If the remote host responds with a SYN/ACK, then the local TCP stack on the scanning system receives the SYN/ACK, but the outbound SYN packet did not come from the local stack (because we manually crafted it via the raw socket), so the SYN/ACK is not part of a legitimate TCP handshake as far as the stack is concerned. Hence, the scanner's local stack sends a RST back to the target system, because the SYN/ACK appears to be unsolicited. You can stop this behavior on the scanning system by adding the following iptables rule to the `OUTPUT` chain before starting a scan with the command:

```
[ext_scanner]# iptables -I OUTPUT 1 -d target -p tcp
--tcp-flags RST RST -j DROP
```

Nmap uses a raw socket to manually build the TCP SYN packets used within its SYN scan mode (`-sS`), the default scanning mode for privileged users. Because the characteristics of these packets are determined by Nmap directly (without the use of the local TCP stack), they differ significantly from TCP SYN packets that the stack would normally have generated. For example, if we initiate a web session to http://www.google.com with a web browser and use tcpdump to display the SYN packet from our local Linux TCP stack, we see the following.

```
[iptablesfw]# tcpdump -i eth0 -l -nn port 80 tcpdump: verbose output
suppressed, use -v or -vv for full protocol decode listening on eth0, link-type
EN10MB (Ethernet), capture size 96 bytes 11:13:40.255182 IP 71.157.X.X.59603 >
72.14.203.99.80: S 2446075733:2446075733(0)win 5840 <mss 1460,sackOK,timestamp
277196169 0,nop,wscale 2>
```

Displayed above in bold are both the window size and the options portion of the TCP header. The specific values for each are defined by the local TCP stack and are used to negotiate a valid TCP session with the remote host.

Unlike the SYN packets generated by the real TCP stack, Nmap doesn't care about negotiating a real TCP session. The only thing Nmap is interested in is whether the port is open (Nmap receives a SYN/ACK), closed (Nmap receives a RST/ACK), or filtered (Nmap receives nothing) on the remote host. Hence, the TCP SYN packet that Nmap puts on the wire just needs to qualify to the remote host as a TCP packet with the SYN flag set so that the remote TCP stack either responds with a SYN/ACK, a RST/ACK, or nothing (if the port is filtered).

For versions of Nmap in the 3.*x* series, no TCP options are included within SYN packets used to scan remote systems, as shown below. (If options were included in the packet, then they would appear after the TCP window size, as shown here in bold.)

```
11:17:30.313099 IP 71.157.X.X.52831 > 72.14.203.99.80: S
2001815651:2001815651(0) win 3072
```

For recent versions of Nmap, the Maximum Segment Size (MSS) value is included within SYN packets that it sends, as shown below in bold.

```
15:55:57.521882 IP 71.157.X.X.58302 > 72.14.203.99.80: S 197554866:197554866(0)
win 2048 <mss 1460>
```

If we run a SYN scan now against the iptablesfw system, the same ports that we saw from the `connect()` scan are reported as open, but there are fewer TCP options than for the `connect()` scan, as you can see. That is, the options string for the SYN scan is `020405B4` whereas the options string for the `connect()` scan in the previous section is `020405B40402080A36295772000000001030306`.

```
[ext_scanner]# nmap -P0 -sS 71.157.X.X Starting Nmap 4.01 (
http://www.insecure.org/nmap/ ) at 2007-07-03 00:27 EDT
Interesting ports on 71.157.X.X: (The 1670 ports scanned but not
shown below are in state: filtered) PORT STATE SERVICE 80/tcp
open http 443/tcp open https Nmap finished: 1 IP address (1 host
up) scanned in 22.334 seconds

[iptablesfw]# grep SYN /var/log/messages | tail -n 1 Jul 3
00:27:59 iptablesfw kernel: DROP IN=eth0 OUT=
MAC=00:13:d3:38:b6:e4:00:30:48:80:4e:37:08:00 SRC=144.202.X.X
DST=71.157.X.X LEN=44 TOS=0x00 PREC=0x00 TTL=52 ID=21049
PROTO=TCP SPT=43996 DPT=658 WINDOW=1024 RES=0x00 SYN URGP=0 OPT
```

```
(020405B4)
```

TCP FIN, XMAS, and NULL Scans The FIN, XMAS, and NULL scans operate

on the principle that any TCP stack (that adheres to the RFC) should respond in a particular way if a surprise TCP packet that does not set the SYN, ACK, or RST control bits is received on a port. If the port is closed, then TCP responds with a RST/ACK, but if the port is open, TCP does not respond with *any* packet at all.

The following example shows a FIN scan of the iptablesfw system, and note at that all ports are reported as `open|filtered` by Nmap. Because a surprise FIN packet is not part of any legitimate TCP connection, all of the FIN packets (even those to open ports) are matched against the `INVALID` state rule in the iptables policy and subsequently logged and dropped. (See the `DROP INVALID` log prefix at and the FIN flag set at below.)

```
[ext_scannner]# nmap -P0 -sF 71.157.X.X Starting Nmap 4.01 (
http://www.insecure.org/nmap/ ) at 2007-07-03 00:33 EDT All 1672
scanned ports on 71.157.X.X are: open|filtered Nmap finished: 1
IP address (1 host up) scanned in 36.199 seconds

[iptablesfw]# grep FIN /var/log/messages | tail -n 1 Jul 3
00:34:17 iptablesfw kernel: DROP INVALID IN=eth0 OUT=
MAC=00:13:d3:38:b6:e4:00:30:48:80:4e:37:08:00 SRC=144.202.X.X
DST=71.157.X.X LEN=40 TOS=0x00 PREC=0x00 TTL=54 ID=50009
PROTO=TCP SPT=60097 DPT=1437 WINDOW=3072 RES=0x00 FIN URGP=0
```

TCP ACK Scans The TCP ACK scan (Nmap `-sA`) sends a TCP ACK packet to each scanned port and looks for RST packets (not RST/ACK packets, in this case) from both open and closed ports. If no RST packet is returned by a target port, then Nmap infers that the port is filtered, as shown in the example ACK scan against the iptablesfw system below at .

The goal of the ACK scan is not to determine whether a port is open or closed, but whether a port is filtered by a stateful firewall. Because the iptables firewall is stateful whenever the Netfilter connection tracking subsystem is used (via the state match), no surprise ACK packets make it into the TCP stack on the iptablesfw system. Therefore, as shown here, no RST packets are returned to the scanner (note the ACK flag set at):

```
[ext_scanner]# nmap -P0 -sA 71.157.X.X Starting Nmap 4.01 (
http://www.insecure.org/nmap/ ) at 2007-07-03 00:36 EDT All 1672
scanned ports on 71.157.X.X are: filtered Nmap finished: 1 IP
address (1 host up) scanned in 36.191 seconds [iptablesfw]# grep
ACK /var/log/messages | tail -n 1 Jul 3 00:37:18 iptablesfw
kernel: DROP IN=eth0 OUT=
MAC=00:13:d3:38:b6:e4:00:30:48:80:4e:37:08:00 SRC=144.202.X.X
DST=71.157.X.X LEN=40 TOS=0x00 PREC=0x00 TTL=43 ID=51322
PROTO=TCP SPT=62068 DPT=6006 WINDOW=4096 RES=0x00 ACK URGP=0
```